




Юрий Магда

# Ассемблер для процессоров Intel Pentium

- Базовая программная архитектура процессоров
- Технологии параллельной обработки данных
- Архитектурно-программные решения для Intel Pentium
- Примеры программного кода




ПИТЕР




Юрий Магда

# Ассемблер для процессоров Intel Pentium




Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2006

ББК 32.973-018.1
УДК 004.43
М12

**Магда Ю. С.**
М12 Ассемблер для процессоров Intel Pentium. — СПб.: Питер, 2006. — 410 с.: ил.

ISBN 5-469-00662-X

Издание посвящено вопросам программирования на языке ассемблера для процессоров Intel Pentium. Рассмотрен широкий круг вопросов, начиная с основ программирования на ассемблере и заканчивая применением самых современных технологий обработки данных, таких как MMX, SSE и SSE2. Материал книги раскрывает методику оптимизации программного кода для всех поколений процессоров Intel Pentium, включая Intel Pentium 4. Теоретический материал подкреплен многочисленными примерами программного кода. Для широкого круга читателей, от студентов до опытных разработчиков программного обеспечения.

ББК 32.973-018.1
УДК 004.43



ISBN 5-469-00662-X 

# Краткое содержание

Введение . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
**Глава 1.** Базовая архитектура процессоров Intel x86 . . . . . . . . . . . . . . . . 14
**Глава 2.** Основы создания приложений на языке ассемблера . . . . . . . . . . 21
**Глава 3.** Синтаксис языка ассемблера . . . . . . . . . . . . . . . . . . . . . . . 26
**Глава 4.** Структура программы на языке ассемблера . . . . . . . . . . . . . . . 53
**Глава 5.** Организация вычислительных циклов . . . . . . . . . . . . . . . . . . 61
**Глава 6.** Процедуры на языке ассемблера . . . . . . . . . . . . . . . . . . . . . 93
**Глава 7.** Операции со строками и массивами . . . . . . . . . . . . . . . . . . . 120
**Глава 8.** Арифметические и логические операции . . . . . . . . . . . . . . . . 163
**Глава 9.** Использование математического сопроцессора . . . . . . . . . . . . 206
**Глава 10.** Интерфейс с языками высокого уровня . . . . . . . . . . . . . . . . . 250
**Глава 11.** Процессоры Intel Pentium в современных разработках . . . . . . . . 266
**Глава 12.** MMX-расширение процессоров Intel Pentium . . . . . . . . . . . . . . 270
**Глава 13.** SSE-расширение процессоров Intel Pentium . . . . . . . . . . . . . . 318
**Глава 14.** Технология SSE2 в процессорах Intel Pentium 4 . . . . . . . . . . . . 362
Заключение . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 397
**Приложение А.** Базовые инструкции процессоров 80x86 . . . . . . . . . . . . . 399
**Приложение Б.** Специальные инструкции процессоров 80x86 . . . . . . . . . . 406
Список литературы . . . . . . . . . . . . . . . . . . . . . . . . . . . 409

# Содержание

Введение . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
Структура книги . . . . . . . . . . . . . . . . . . . . . . . . . 11
От издательства . . . . . . . . . . . . . . . . . . . . . . . . . 13

**Глава 1.** Базовая архитектура процессоров Intel x86 . . . . . . . . . . 14

**Глава 2.** Основы создания приложений на языке ассемблера . . . . . . . 21
2.1. Ассемблирование исходного текста . . . . . . . . . . . . . . . . 23
2.2. Компоновка программ . . . . . . . . . . . . . . . . . . . . . . 23

**Глава 3.** Синтаксис языка ассемблера . . . . . . . . . . . . . . . . 26
3.1. Представление данных в компьютере . . . . . . . . . . . . . . . 26
3.2. Первичные элементы языка ассемблера . . . . . . . . . . . . . . 32
3.3. Программная модель процессора Intel Pentium . . . . . . . . . . 38

**Глава 4.** Структура программы на языке ассемблера . . . . . . . . . . 53
4.1. Организация сегментов . . . . . . . . . . . . . . . . . . . . . 53
4.2. Директивы управления сегментами и моделями памяти макроассемблера MASM . . . . . . . . . . . . . . . . . . . . . . 54
4.3. Структура программ на ассемблере MASM . . . . . . . . . . . . . 57

**Глава 5.** Организация вычислительных циклов . . . . . . . . . . . . . 61
5.1. Условные переходы и ветвления . . . . . . . . . . . . . . . . . 63
5.2. Команда безусловного перехода jmp . . . . . . . . . . . . . . . 66
5.3. Организация циклов . . . . . . . . . . . . . . . . . . . . . . . 72
5.4. Оптимизация кода в процессорах Intel Pentium . . . . . . . . . . 78

**Глава 6.** Процедуры на языке ассемблера . . . . . . . . . . . . . . . 93
6.1. Организация стека . . . . . . . . . . . . . . . . . . . . . . . 94
6.2. Принципы организации подпрограмм . . . . . . . . . . . . . . . 100

6.3. Параметры процедур и возвращаемые значения . . . . . . . . . . . . . . 110
6.4. Использование общих переменных в процедурах . . . . . . . . . . . . . 116

**Глава 7.** Операции со строками и массивами . . . . . . . . . . . . . . . 120
7.1. Пересылка и копирование данных . . . . . . . . . . . . . . . . . . . . 124
7.2. Сравнение строк и массивов . . . . . . . . . . . . . . . . . . . . . . . 135
7.3. Сканирование строк и массивов . . . . . . . . . . . . . . . . . . . . . 145
7.4. Использование команд lods и stos . . . . . . . . . . . . . . . . . . . 150
7.5. Массивы строк . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 154
7.6. Полезные алгоритмы . . . . . . . . . . . . . . . . . . . . . . . . . . 157
7.7. Полезные советы . . . . . . . . . . . . . . . . . . . . . . . . . . . . 160

**Глава 8.** Арифметические и логические операции . . . . . . . . . . . . 163
8.1. Логические команды . . . . . . . . . . . . . . . . . . . . . . . . . . 163
8.2. Команды сканирования битов . . . . . . . . . . . . . . . . . . . . . . 166
8.3. Команды сдвига и циклического сдвига . . . . . . . . . . . . . . . . . 168
8.4. Обработка целых чисел . . . . . . . . . . . . . . . . . . . . . . . . . 170
8.5. Обработка данных в форматах ASCII и BCD . . . . . . . . . . . . . . . 189
8.6. Преобразование ASCII-чисел в двоичный формат . . . . . . . . . . . . 197
8.7. Преобразование двоичных чисел в формат ASCII . . . . . . . . . . . . 199
8.8. Полезные алгоритмы и программы . . . . . . . . . . . . . . . . . . . 200

**Глава 9.** Использование математического сопроцессора . . . . . . . . . 206
9.1. Типы данных сопроцессора . . . . . . . . . . . . . . . . . . . . . . . 207
9.2. Архитектура сопроцессора . . . . . . . . . . . . . . . . . . . . . . . 209
9.3. Система команд математического сопроцессора . . . . . . . . . . . . 214

**Глава 10.** Интерфейс с языками высокого уровня . . . . . . . . . . . . 250
10.1. Общие принципы построения интерфейсов . . . . . . . . . . . . . . . 250
10.2. Интерфейс ассемблерных процедур с Delphi 2005 . . . . . . . . . . . 255
10.3. Интерфейс ассемблерных процедур с Visual C++ .NET 2005 . . . . . . 261

**Глава 11.** Процессоры Intel Pentium в современных разработках . . . . 266
11.1. Микроархитектура Intel NetBurst . . . . . . . . . . . . . . . . . . . 266
11.2. Особенности работы приложений с процессором Intel Pentium 4 . . . . 268

**Глава 12.** MMX-расширение процессоров Intel Pentium . . . . . . . . . 270
12.1. Команды передачи данных . . . . . . . . . . . . . . . . . . . . . . . 274
12.2. Команды сложения . . . . . . . . . . . . . . . . . . . . . . . . . . . 275
12.3. Команды вычитания . . . . . . . . . . . . . . . . . . . . . . . . . . 285
12.4. Команды упаковки и распаковки данных . . . . . . . . . . . . . . . . 287
12.5. Команды умножения . . . . . . . . . . . . . . . . . . . . . . . . . . 302
12.6. Команды сравнения . . . . . . . . . . . . . . . . . . . . . . . . . . 307
12.7. Логические команды . . . . . . . . . . . . . . . . . . . . . . . . . . 311
12.8. Команды сдвига . . . . . . . . . . . . . . . . . . . . . . . . . . . . 313
12.9. Дополнительные команды . . . . . . . . . . . . . . . . . . . . . . . 316

**Глава 13.** SSE-расширение процессоров Intel Pentium . . . . . . . . . . 318
13.1. Команды передачи данных . . . . . . . . . . . . . . . . . . . . . . . . . . 322
13.2. Арифметические команды . . . . . . . . . . . . . . . . . . . . . . . . . . 328
13.3. Команды сравнения . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 341
13.4. Команды преобразования . . . . . . . . . . . . . . . . . . . . . . . . . . 347
13.5. Логические команды . . . . . . . . . . . . . . . . . . . . . . . . . . . . 354
13.6. Команды управления состоянием . . . . . . . . . . . . . . . . . . . . . . 356
13.7. Команды распаковки данных . . . . . . . . . . . . . . . . . . . . . . . . 356
13.8. Команды управления кэшированием . . . . . . . . . . . . . . . . . . . . . 360

**Глава 14.** Технология SSE2 в процессорах Intel Pentium 4 . . . . . . . . 362
14.1. Команды обработки 128-разрядных данных с плавающей точкой . . . . . 364
14.2. Команды обработки 128-разрядных целочисленных данных . . . . . . . . 385

Заключение . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 397

**Приложение А.** Базовые инструкции процессоров 80x86 . . . . . . . . . . 399

**Приложение Б.** Специальные инструкции процессоров 80x86 . . . . . . . 406

Список литературы . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 409

# Введение

Эта книга посвящена описанию возможностей языка ассемблера процессоров Intel Pentium. Книга не является учебником по языку ассемблера, хотя и может использоваться в этом качестве, — скорее это расширенное руководство по применению ассемблера процессоров Intel Pentium. Материал книги содержит много справочной информации по командам ассемблера и современным технологиям обработки данных. Изучение современного ассемблера — задача далеко не простая, и эта книга позволит читателю успешно ее решить.

Язык ассемблера появился вместе с появлением процессоров и тесно связан с их архитектурой, позволяя напрямую обращаться к аппаратным ресурсам компьютера. Часто у читателей возникает вопрос: а зачем вообще нужно изучать язык ассемблера, когда имеются развитые средства программирования на языках высокого уровня, такие, например, как Visual C++ .NET фирмы Microsoft или Borland Delphi 2005? Тем более что помимо этих средств есть еще целый спектр специализированных программных продуктов для разработки офисных приложений, баз данных, электронных таблиц и т. д. Подобные программы называются средствами быстрой разработки и позволяют в считанные недели создавать самые сложные приложения.

Тем не менее значение языка ассемблера трудно переоценить. Все без исключения средства разработки программ в той или иной степени используют ассемблер. К примеру, большинство библиотечных функций языков C++ и Pascal, на основе которых построены такие мощные инструменты разработки, как Visual C++ и Delphi, написаны на ассемблере. Мультимедийные приложения, программы обработки сигналов и многие другие используют высокопроизводительные библиотеки функций, разработанные с помощью ассемблерных команд технологии SIMD. Наконец, если требуется, чтобы приложение работало максимально быстро и занимало меньше памяти (а это нужно для встроенных и мобильных систем в различных отраслях промышленности), то применение ассемблера является едва ли не единственным способом достижения цели. По этой причине большинство

приложений, работающих в режиме реального времени, либо написаны целиком на ассемблере, либо используют в критических участках кода ассемблерный код.

Даже по этим нескольким примерам видно, что язык ассемблера имеет свои сферы применения, свои ниши, которые никогда и ничем не будут заняты. Кроме того, как уже отмечалось, при разработке приложений на языках высокого уровня критические секции, требующие высокой скорости выполнения, пишутся на ассемблере. Именно поэтому в Visual C++ .NET и Delphi 2005 имеется возможность создавать программный код на встроенном ассемблере. Должен заметить, что фирма Microsoft постоянно совершенствует встроенный ассемблер.

Вряд ли кому-то придет в голову разрабатывать большие и многофункциональные приложения на языке ассемблера, но ускорить производительность работы таких приложений с помощью ассемблера можно. По сравнению с языками высокого уровня ассемблер обладает одним фундаментальным преимуществом, проистекающим из его природы, — он позволяет писать самый быстрый и компактный код. Изучение языка ассемблера дает программисту одно очень важное преимущество — он глубже начинает понимать принципы работы приложений, написанных на любых языках, в том числе и на языках высокого уровня. Ассемблер очень помогает при разработке программ на языках высокого уровня, поскольку знание низкоуровневого программирования позволяет выбирать оптимальные решения.

Что же касается инструментальных средств для разработки приложений на «чистом» ассемблере, то в последнее время появились очень мощные приложения такого рода, что вынуждает по-другому взглянуть на проблему. Из таких инструментальных средств проектирования можно выделить в первую очередь макроассемблер MASM32, а также AsmStudio и NASM. Эти и другие инструменты разработки программ имеют самый современный графический интерфейс. Не следует забывать и о том, что для ассемблера разработаны многочисленные библиотеки функций, приближающие этот язык по своим функциональным возможностям к высокоуровневым средствам разработки приложений.

Материал книги охватывает полный спектр архитектурно-программных решений для процессоров Intel Pentium, включая как базовую программную архитектуру и набор основных команд ассемблера, так и современные технологии параллельной обработки данных (SIMD). Эта книга задумана как расширенный справочник по применению ассемблера в практических разработках, хотя может быть использована и как практическое пособие для программистов-разработчиков, желающих углубить свои знания о современных технологиях программирования на ассемблере.

Материал книги включает много примеров программного кода, в том числе и для технологий SIMD. Этими практическими примерами подкрепляются большинство теоретических аспектов, рассматриваемых в книге. По мнению автора, такой путь является наиболее эффективным для изучения языка ассемблера.

Все примеры программ являются полностью работоспособными и проверены автором. Они демонстрируют ключевые моменты использования тех или иных команд или технологий и реализованы в виде коротких процедур. Такая методика выбрана сознательно, поскольку длинные и сложные программы обычно

запутывают читателя, и при их анализе легко теряются ключевые моменты, ради которых эти программы, собственно, и были разработаны. Любой пример несложно адаптировать для дальнейшего использования в собственных разработках.

Для разработки примеров используется макроассемблер MASM фирмы Microsoft с компилятором версии 7.10*xxxx*. Этот компилятор включен в состав Windows XP DDK и Windows Server 2003 DDK. Подойдет и компилятор версии 6.14*xxxx*, но в этом случае примеры применения технологий SIMD компилировать будет невозможно. В качестве среды разработки можно порекомендовать свободно распространяемый макроассемблер MASM32 версии 8, который включает в себя компилятор ML версии 6.14*xxxx* и компоновщик LINK версии 5.12*xxxx* фирмы Microsoft.

Во всех примерах синтаксис языка ассемблера максимально упрощен, используется минимум высокоуровневых конструкций языка. В книге не приводится детальное описание компилятора MASM, а упоминаются лишь те сведения, которые необходимы для работы.

Книга рассчитана на широкий круг читателей — от начинающих программистов до опытных разработчиков.

# Структура книги

Структура книги такова, что материал можно изучать выборочно по отдельным главам или последовательно, начиная с первой главы. Это позволяет различным категориям читателей изучать тот материал, который им более всего интересен.

Книга состоит из 14 глав.

- Глава 1, «Базовая архитектура процессоров Intel x86». В этой главе рассматриваются базовая архитектура процессоров x86 фирмы Intel и эволюция к последним моделям процессоров Intel Pentium.
- Глава 2, «Основы создания приложений на языке ассемблера». Материал этой главы посвящен общим принципам создания программ на ассемблере. Здесь также рассмотрены основные этапы компиляции и компоновки приложений с использованием макроассемблера MASM фирмы Microsoft.
- Глава 3, «Синтаксис языка ассемблера». В этой главе проанализирован синтаксис языка ассемблера, включая основные типы данных, модели памяти и типы адресации при работе с процессорами Intel.
- Глава 4, «Структура программы на языке ассемблера». В этой главе проанализирована сегментная структура ассемблерных программ и ее взаимосвязь с используемыми моделями памяти.
- Глава 5, «Организация вычислительных циклов». Материал главы посвящен организации вычислительных алгоритмов с использованием команд условных и безусловных переходов. Здесь также рассматриваются варианты оптимизации ветвлений в программах с применением специальных команд процессоров Intel Pentium.

- Глава 6, «Процедуры на языке ассемблера». В этой главе описаны процесс разработки и применения процедур на языке ассемблера, а также вопросы организации и использования стека для передачи параметров. Рассмотрены различные варианты обработки данных в процедурах, обращений к регистрам и памяти.
- Глава 7, «Операции со строками и массивами». Здесь рассматриваются строковые команды процессора Intel Pentium и практические аспекты их применения при обработке символьных строк и числовых массивов. Проанализированы методы оптимизации строковых операций.
- Глава 8, «Арифметические и логические операции». Материал главы посвящен анализу арифметических и логических команд процессора, а также преобразованиям целочисленных данных из одних форматов в другие.
- Глава 9, «Использование математического сопроцессора». Здесь рассматриваются вопросы применения математического сопроцессора в операциях над числами с плавающей точкой и способы создания эффективных алгоритмов обработки данных.
- Глава 10, «Интерфейс с языками высокого уровня». Материал главы посвящен применению отдельно скомпилированных ассемблерных модулей в программах на языках высокого уровня. В главе подробно анализируются методы передачи параметров в процедуры и получения результатов.
- Глава 11, «Процессоры Intel Pentium в современных разработках». В главе рассматриваются общие вопросы применения процессоров последних поколений Intel Pentium 4 в разработке высокоэффективных приложений. Показаны возможности оптимизации приложений для процессоров Pentium 4.
- Глава 12, «MMX-расширение процессоров Intel Pentium». Здесь проанализированы основные аспекты использования технологии MMX для повышения производительности мультимедийных приложений и операций с целыми числами.
- Глава 13, «SSE-расширение процессоров Intel Pentium». В главе рассматриваются основные аспекты применения технологии SSE для повышения производительности операций с плавающей точкой в коротком формате и возможности оптимизации программ.
- Глава 14, «Технология SSE2 в процессорах Intel Pentium 4». Глава посвящена вопросам применения технологии SSE2 для повышения производительности операций с плавающей точкой двойной точности. Материал сопровождается многочисленными примерами практического применения данной технологии.

Материал книги дополнен справочником по системе команд процессоров Intel (Приложение А). Поскольку полная система команд насчитывает несколько сотен наименований, приведены только наиболее часто используемые команды.

Автор благодарит коллектив издательства «Питер» за помощь в подготовке книги к изданию. Особая признательность жене Юлии за поддержку и помощь в написании книги.

# От издательства

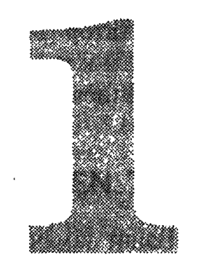
Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты: comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы можете найти по адресу http://www.piter.com/download.

Подробную информацию о наших книгах вы найдете на веб-сайте издательства: http://www.piter.com.

# Базовая архитектура процессоров Intel x86



Успешное применение языка ассемблера невозможно без знания архитектуры процессоров Intel. Процессоры Intel в настоящее время доминируют на рынке, и многие архитектурные решения, на основе которых они построены, в той или иной степени используются и другими производителями процессоров. Поскольку все современные процессоры Intel базируются на архитектуре 8086, то обычно говорят об архитектуре Intel x86.

Вкратце рассмотрим эволюцию процессоров фирмы Intel. В 1979 г. фирма Intel первой выпустила 16-разрядный микропроцессор 8086, возможности которого были близки к возможностям процессоров мини-компьютеров 70-х годов. Микропроцессор 8086 стал базовым для целого семейства процессоров, которое называют семейством 80x86 или x86.

Чуть позже появился процессор 8088, архитектурно совместимый с процессором 8086 и имеющий 16-разрядные регистры, но оперирующий с внешними данными размером в 8 бит. В 1981 г. появились процессоры 80186/80188, наследующие базовую архитектуру процессоров 8086, но обладающие дополнительными возможностями. Это поколение включало дополнительные аппаратно-программные компоненты: контроллер прямого доступа к памяти, счетчик/таймер и контроллер прерываний. Кроме того, система команд этих процессоров была расширена. Несмотря на это, широкого распространения данные процессоры не получили.

Следующим этапом в разработке новых идей стал процессор 80286. В этой модели были использованы новые подходы, которые применялись в микрокомпьютерах и больших компьютерах. Процессор 80286 мог работать в двух режимах: в режиме реальных адресов (эмуляция процессора 8086) и в защищенном режиме виртуальных адресов (protected virtual address mode), который предоставлял новые возможности для программистов. В этом режиме можно было работать с расширенным адресным пространством памяти размером в 16 Мбайт, также поддерживались виртуальная память и мультизадачность.

Новый 32-разрядный процессор 80386 позволил успешно решить две основные задачи: он обеспечивал совместимость с предыдущими поколениями процессоров и одновременно повышал производительность выполнения программ. Совместимость с процессорами 8086 достигалась за счет включения в аппаратно-программную архитектуру режима реальной адресации (real address mode).

В этом режиме процессор 80386 мог выполнять 16-разрядный программный код процессора 80286 без каких-либо ограничений. В этом же режиме он мог запускать 32-разрядные программы, что повышало производительность системы. В 32-разрядном режиме были реализованы новые возможности процессора 80386: масштабированная индексная адресация памяти, ортогональное использование регистров общего назначения, новые команды и средства отладки. Адресное пространство памяти позволяло работать с 4 Гбайт данных.

По сравнению с предыдущими поколениями процессоров процессор 80386 обеспечивал большее быстродействие (3–4 миллиона операций в секунду) и возможность работы со страничной виртуальной памятью.

В 1989 г. фирма Intel выпустила процессор i486, содержащий более миллиона транзисторов в чипе. Являясь полностью программно совместимым с процессорами 386, он предоставлял новые возможности по обеспечению многозадачности систем и многоуровневого кэширования. Встроенная система тестирования позволяла проверять работоспособность аппаратной логики, кэш-памяти и аппаратного постраничного преобразования адресов памяти. Отладочные средства обеспечивали установку ловушек контрольных точек в исполняемом коде и во время доступа к данным. Процессор i486 имел встроенный аппаратный кэш для хранения 8 Кбайт команд и данных, что увеличивало быстродействие системы, одновременно уменьшая степень использования процессором внешней шины.

Следующим шагом в повышении производительности компьютерных систем стало появление процессоров Intel Pentium. По сравнению с процессором i486 был добавлен второй конвейер команд, что дало более высокую скорость выполнения команд. Оба конвейера команд, обозначаемые **u** и **v**, при совместной работе обеспечивают выполнение двух инструкций процессора за один машинный цикл.

Размер встроенного кэша первого уровня увеличен в два раза, при этом для команд и данных используется по 8 Кбайт памяти. В процессоре применяются более эффективные по сравнению с i486 алгоритмы прямой (write-through) и обратной записи (write-back).

В процессорах Intel Pentium впервые был использован так называемый алгоритм прогнозирования программных ветвлений и циклов (branch prediction). С помощью такого алгоритма обеспечивается более эффективное управление потоком команд программы. Адреса прогнозируемых переходов хранятся в аппаратно реализованной таблице ветвлений. Кроме этого, в Intel Pentium были внесены аппаратные расширения, позволяющие более эффективно работать в режиме виртуального процессора 8086 с адресным пространством в 4 Мбайт и размером страницы 4 Кбайт.

Основные регистры процессора остались 32-разрядными, но были добавлены внутренние шины передачи данных размерностью в 128 и 256 бит, что обеспечивает

более быстрый обмен данными внутри процессора. Кроме того, внешняя шина данных в процессоре позволяет работать с 64-разрядными данными.

В процессоре Intel Pentium сочетаются высокая производительность, совместимость, интеграция данных и наращиваемость. Это достигается за счет того, что процессор обладает:

- суперскалярной архитектурой;
- раздельным кэшированием программного кода и данных;
- блоком прогнозирования адреса перехода;
- высокопроизводительным блоком операций с плавающей точкой;
- расширенной 64-разрядной шиной данных;
- поддержкой многопроцессорного режима работы;
- средствами задания размера страницы памяти;
- средствами обнаружения ошибок и функциональной избыточности;
- возможностями для управления производительностью.

Совместимая только с Intel суперскалярная двухконвейерная промышленная архитектура процессора Intel Pentium позволяет ему достичь нового уровня производительности посредством выполнения более чем одной команды за один такт. Термин «суперскалярная» обозначает процессорную архитектуру, которая содержит более одного вычислительного блока. Все эти вычислительные блоки, или конвейеры, являются узлами, где происходят все основные процессы обработки данных и команд. Они позволяют выполнить значительно большее число команд за одно и то же процессорное время по сравнению с предыдущими поколениями процессоров.

Появление суперскалярной архитектуры процессора Intel Pentium представляет собой естественное развитие предыдущего семейства процессоров с 32-разрядной архитектурой фирмы Intel. Например, процессор i486 способен выполнять несколько своих команд за один такт, в то время как предыдущие семейства процессоров фирмы Intel требовали множества циклов тактовой частоты для выполнения одной команды.

Другим важным усовершенствованием, реализованным в процессорах Pentium, является раздельное кэширование. Кэширование повышает производительность посредством активизации места временного хранения часто используемых программного кода и данных, получаемых из быстрой памяти, заменяя по возможности обращение к внешней системной памяти для некоторых команд.

Раздельное кэширование позволяет выполнять несколько команд одновременно. Кэш-память программного кода и данных процессора Pentium содержит по 8 Кбайт информации и организована как набор двухканального ассоциативного кэша. Такая кэш-память предназначена для записи только предварительно просмотренного 32-байтового сегмента, причем работает быстрее, чем внешний кэш. Все это потребовало использования 64-разрядной внутренней шины данных, которая обеспечивает возможность двойного кэширования и суперскалярной конвейерной обработки одновременно с загрузкой последующих данных.

Кэш данных имеет два интерфейса, по одному для каждого из конвейеров, что позволяет ему обеспечивать данными две отдельные инструкции в течение одного машинного цикла. После того как данные извлекаются из кэша, они записываются в основную память в режиме обратной записи. Подобная техника кэширования дает лучшую производительность по сравнению с простым кэшированием с непосредственной записью, при котором процессор записывает данные одновременно в кэш и в основную память. Тем не менее процессор Intel Pentium способен динамически конфигурироваться для поддержки кэширования с непосредственной записью.

Блок прогнозирования адреса перехода позволяет повысить производительность, предварительно определив правильный набор выполняемых команд и полностью заполнив ими конвейеры.

Процессор Intel Pentium дает возможность выполнять математические вычисления на более высоком уровне благодаря использованию усовершенствованного встроенного блока операций с плавающей точкой, который включает в себя 8-тактовый конвейер и аппаратную реализацию основных математических функций. 4-тактовые конвейерные команды для операций с плавающей точкой дополняют 4-тактовую целочисленную конвейеризацию. Большая часть команд, оперирующих данными с плавающей точкой, может выполняться в одном целочисленном конвейере, после чего помещается в конвейер операций с плавающей точкой. Обычные операции с плавающей точкой, такие, как сложение, умножение и деление, реализованы аппаратно для ускорения процесса вычислений.

В результате этих усовершенствований процессор Intel Pentium выполняет команды для операций с плавающей точкой в пять раз быстрее, чем работающий на частоте 33 МГц процессор i486, оптимизируя их для высокоскоростных вычислений в мультимедийных приложениях, а также в 3D- и CAD/CAM-приложениях.

К числу аппаратных нововведений следует отнести и более совершенный программируемый контроллер прерываний (Advanced Programmable Interrupt Controller, APIC), позволяющий создавать системы с несколькими процессорами Intel Pentium. Но самым радикальным усовершенствованием процессоров Intel Pentium стало внедрение технологии MMX (MultiMedia eXtensions — мультимедийные расширения). В технологии MMX для организации параллельных вычислений над упакованными 64-разрядными целыми числами используется модель SIMD (Single Instruction, Multiple Data — одна команда, много данных). Параллельная обработка целочисленных данных не требует дополнительных регистров процессора — задействуются регистры математического сопроцессора. Технология MMX позволила существенно повысить производительность мультимедийных приложений, программ обработки звука и изображений, программ криптографии и сжатия данных.

Для аппаратуры компьютера процессор Intel Pentium представляет собой 32-разрядное устройство. Внешняя шина данных к памяти является 64-разрядной, что обеспечивает передачу удвоенного объема данных за один цикл шины. Процессор поддерживает несколько типов циклов, включая цикл пакетного режима, в течение которого в кэш данных передается 256-разрядный пакет данных.

Шина данных является главной магистралью, которая передает информацию между процессором и подсистемой памяти. Благодаря 64-разрядной шине данных процессор Intel Pentium существенно повысил скорость передачи — до 528 Мбайт/с для 66 МГц по сравнению со 160 Мбайт/с для 50 МГц процессора i486. Эта расширенная шина данных поддерживает высокоскоростные вычисления за счет одновременной загрузки командами и данными процессорного блока, благодаря чему достигается еще большая общая производительность процессора Intel Pentium по сравнению с процессором i486.

Первые модели процессора Intel Pentium работали на частоте 60 и 66 МГц и обменивались данными с внешней кэш-памятью второго уровня по 64-разрядной шине данных, работающей на тактовой частоте процессорного ядра. Однако здесь есть некоторые сложности. При возрастании скорости процессора Intel Pentium все сложнее становится и дороже обходится его согласование с электронным интерфейсом на материнской плате.

По этой причине быстрые процессоры Intel Pentium используют делитель частоты для синхронизации внешней шины путем задания меньшей частоты. Например, у процессора Intel Pentium с частотой 100 МГц внешняя шина работает на частоте 66 МГц, а у процессора с частотой 90 МГц — на частоте 60 МГц. Процессор задействует одну и ту же шину для доступа к основной памяти и к периферийным подсистемам, таким, как шина PCI.

Дальнейшим усовершенствованием процессоров Intel Pentium стала модель P6, выпущенная в 1995 г. Это поколение процессоров базировалось на суперскалярной архитектуре, что позволило без перехода на другую технологию изготовления кристалла значительно повысить производительность. Первым процессором семейства P6 стал Intel Pentium Pro. Далее были разработаны более совершенные процессоры этой линейки, известные как Intel Pentium II, Intel Pentium II Xeon, Intel Celeron, Intel Pentium III и Intel Pentium III Xeon.

Появление процессора Pentium Pro ознаменовало собой значительный шаг вперед по сравнению с Intel Pentium. Несмотря на то что в процессоре Intel Pentium впервые была реализована суперскалярная форма архитектуры x86, она имела определенные ограничения. В этой архитектуре имеется всего два целочисленных конвейера, которые могут обрабатывать две команды параллельно, но только если они следуют друг за другом — здесь отсутствует алгоритм, позволяющий предсказывать ветвления в программе.

В процессоре Pentium Pro реализована новая модель суперскалярной архитектуры, позволяющая одновременно выполнять пять команд. Внедрение такой архитектуры позволило достичь высокой пропускной способности, но одновременно потребовало значительного улучшения схемы кэширования. По сравнению с обычным процессором Intel Pentium, в Pentium Pro пришлось расширить файл регистров, повысить глубину очереди упреждающей выборки и условного выполнения команд, усовершенствовать алгоритм прогнозирования адресов перехода и реализовать обработку данных не в порядке их поступления, а по мере их готовности.

Суперскалярная архитектура процессора Pentium Pro позволяет выполнять максимум три машинные команды в течение одного цикла. Кроме того, в процессоре используется концепция динамического выполнения команд (изменение порядка выполнения команд, улучшенный алгоритм прогнозирования ветвлений

и опережающего выполнения команд). Суперскалярная архитектура реализована с помощью трех блоков декодирования команд, работающих параллельно. Команды разбиваются на так называемые микрооперации, которые выполняются параллельно в пяти исполнительных модулях: двух целочисленных, двух FPU (Floating-Point Unit) и одном модуле интерфейса памяти.

В программно-аппаратную архитектуру процессора входит и специальный модуль отложенных операций (retirement unit), позволяющий выполнять последовательность микроопераций в нужном порядке даже при наличии ветвлений.

По сравнению с процессором Intel Pentium, в Pentium Pro был увеличен размер кэша второго уровня (2nd-level cache) до 256 Кбайт. Процессор Pentium Pro имеет 36-разрядную адресную шину, позволяющую расширить пространство физических адресов до 64 Гбайт.

В процессоры Pentium Pro встроена вторичная кэш-память, соединенная с центральным процессором отдельной шиной. Эта кэш-память, представляющая собой статическое (static) оперативное запоминающее устройство (Random Access Memory, RAM) емкостью 256 или 512 Кбайт, значительно повышает производительность вычислительных систем на основе Pentium Pro.

В процессоры Intel Pentium II семейства P6 была включена поддержка технологии MMX. Что касается процессоров Pentium II Xeon, то в них были сконцентрированы все преимущества предыдущих поколений процессоров Intel. Это поколение процессоров было разработано с 4- и 8-кратной масштабируемостью, а также с кэшем второго уровня, имеющим размер 2 Мбайт. Этот процессор предназначается в основном для высокопроизводительных серверов и рабочих станций.

Еще один представитель линейки — процессор Intel Celeron — базируется на архитектуре IA-32 и предназначен для применения в настольных компьютерах. К особенностям этого процессора следует отнести наличие встроенного кэша второго уровня размером 128 Кбайт, а также низкую стоимость.

Значительный шаг вперед был сделан при разработке процессора Intel Pentium III, в котором была реализована технология SSE (Streaming SIMD Extensions — потоковые SIMD-расширения). Эта технология является дальнейшим развитием технологии MMX. В ней используются 128-разрядные регистры для выполнения параллельных операций с упакованными числами с плавающей точкой. Кроме того, в процессорах Pentium III Xeon для повышения производительности имеется улучшенный кэш передачи данных (advanced transfer cache).

Процессор Intel Pentium 4 является последним в линейке процессоров фирмы Intel, базирующихся на архитектуре IA-32, причем здесь была использована микроархитектура NetBurst. В основе этой микроархитектуры лежит оригинальная разработка Intel, позволяющая процессору функционировать на более высокой тактовой частоте, что значительно повышает производительность Pentium 4 по сравнению с предыдущими поколениями процессоров. Микроархитектура NetBurst имеет следующие особенности:

- улучшенное схемотехническое и конструктивное исполнение, обеспечивающее высокую производительность операций (rapid execution engine);
- поддержка технологии Hyper Pipelined;

- поддержка технологии Advanced Dynamic Execution;
- принципиально новая система кэширования команд-данных;
- поддержка технологии SSE2 (Streaming SIMD Extensions 2), которая обеспечивает расширение возможностей технологий MMX и SSE фирмы Intel за счет включения в систему команд 128-разрядной целочисленной арифметики и 128-разрядной арифметики чисел с плавающей точкой двойной точности;
- гибкая система управления кэшированием данных и памятью.

Аппаратно микроархитектура NetBurst реализована в виде быстродействующей (400 МГц) системной шины, обладающей впечатляющими возможностями по обработке данных:

- производительность операций — до 3,2 Гбайт/с, что более чем в 3 раза превышает производительность Pentium III;
- тактовая частота шины — 100 МГц с возможностью учетверения скорости (400 МГц);
- высокая степень конвейеризации транзакций;
- возможность доступа к 128-разрядным данным посредством 64-разрядных элементов;
- совместимость с существующим программным обеспечением и операционными системами, разработанными для архитектуры IA-32.

# Основы создания приложений на языке ассемблера



Материал этой главы посвящен основам создания программ на языке ассемблера, известном также как язык символического кодирования. Язык ассемблера является одним из самых сложных, поскольку требует знания аппаратно-программной архитектуры процессора и особенностей его функционирования. Сложная внутренняя структура, разнообразные форматы команд, многочисленные режимы адресации процессоров Intel x86 ограничивают возможности разработки сложных и объемных программ на языке ассемблера.

Если же требуется разработать небольшое быстрое приложение, потребляющее мало ресурсов, то язык ассемблера является очень серьезной альтернативой языкам высокого уровня. Кроме того, в любых сложных программах всегда существуют критические секции, требующие интенсивных и быстрых вычислений, которые придется разрабатывать не на языке высокого уровня, а на ассемблере.

Для разработки программ на ассемблере создана масса инструментальных средств, но мы будем использовать макроассемблер MASM фирмы Microsoft, включающий в себя несколько утилит. Выбор макроассемблера MASM в качестве среды разработки сделан исходя из следующих соображений:

- MASM является наиболее популярной средой программирования на ассемблере;
- последние версии макроассемблера MASM (7.10*xxxx*) позволяют работать с мультимедийными расширениями (SIMD), которые поддерживаются последними поколениями процессоров. Это является очень важным фактором, поскольку очень мало компиляторов ассемблера поддерживают эти технологии;
- соглашения и форматы файлов, принятые в MASM, поддерживаются большинством компиляторов языка ассемблера;

- стандарты и соглашения, принятые в MASM, полностью совместимы с теми, что приняты в наиболее популярных средах разработки (Microsoft Visual C++ .NET и Borland Delphi 2005). Это свойство позволяет включать скомпилированные макроассемблером объектные файлы в программы, разработанные на языках высокого уровня.

Популярный компилятор TASM, к сожалению, более не поддерживается, и его развитие закончилось несколько лет назад. Серьезным недостатком этого компилятора при всей его привлекательности является невозможность работы с современными архитектурами процессоров Intel, поддерживающими технологии параллельной обработки данных (SIMD).

Хочу уточнить, что мы будем создавать и анализировать программы и процедуры с использованием компилятора ассемблера версий не ниже 6.14*.xxxx*, а при рассмотрении технологий SIMD —не ниже 7.10*.xxxx*. Очень удобен для этих целей свободно распространяемый пакет программ MASM32, содержащий помимо компилятора версии 6.14 также редактор исходных текстов и несколько полезных утилит. На момент написания книги текущей версией MASM32 является 8.2.

Рассмотрим более подробно процесс ассемблирования программ с помощью макроассемблера MASM. Должен заметить, что ассемблирование программ практически не различается для версий 6.14*.xxxx* и выше.

Пакет MASM фирмы Microsoft включает в себя основные программы, необходимые для создания, отладки и сопровождения программ на языке ассемблера. Процесс создания и выполнения программ на языке ассемблера состоит из двух шагов:

1. Ассемблирование (assembling) исходного текста программы в объектный файл. Файл, содержащий исходный текст программы на ассемблере, имеет расширение ASM, а получаемый в результате ассемблирования объектный файл — расширение OBJ.
2. Компоновка полученного объектного файла вместе с другими объектными файлами и/или библиотеками в исполняемый файл (с расширением EXE).

Ассемблирование исходного текста выполняет утилита ml.exe, входящая в состав макроассемблера, а сборку всей программы — утилита link.exe. Эти утилиты могут как выполняться из командной строки, так и включаться в состав программных сценариев. Как ml, так и link принимают множество параметров, часть из которых рассматривается далее.

В состав макроассемблера включен ряд других очень полезных программ, но мы сосредоточим внимание только на утилитах ml и link, поскольку именно они будут использованы при разработке программ и процедур в этой книге. Читатели, заинтересованные в более глубоком изучении возможностей макроассемблера MASM, легко обнаружат полные описания всех программ пакета в Интернете.

После того как исходный текст ассемблерной программы разработан, можно указать специальные условия для его ассемблирования при помощи директивы OPTION. Эта директива имеет несколько параметров, позволяющих управлять процессом ассемблирования программы.

Проанализируем более подробно каждый из этапов создания программ на языке ассемблера.

# 2.1. Ассемблирование исходного текста

Программа ml.exe выполняет два последовательных действия при создании исполняемого файла программы. Во-первых, она обеспечивает трансляцию исходного текста программы в промежуточный объектный файл. Во-вторых, ml.exe вызывает программу link.exe, которая компонует объектные файлы и библиотеки в единую выполняемую программу.

В процессе трансляции исходного текста программы выполняются следующие действия:

1. Анализируются директивы условного ассемблирования, и в случае истинности указанных в них условий выполняются те или иные шаги.
2. Разворачиваются макросы.
3. Вычисляются константные выражения, такие, например, как mydata and 10h, при этом выражения замещаются вычисленными значениями.
4. Декодируются команды и операнды, не находящиеся в памяти. Например, на этом шаге будет декодирована команда mov AX, 10, поскольку она не имеет операндов, расположенных в памяти.
5. Сохраняются смещения переменных в памяти как смещения относительно сегментов, в которых эти переменные расположены.
6. Сегменты и их атрибуты размещаются в объектном файле.
7. В объектном файле сохраняются перемещаемые адреса (relocatable addresses).
8. При необходимости создается файл листинга.
9. Непосредственно программе link.exe передаются некоторые директивы (например, INCLUDELIB и DOSSEG).

Директивы условного ассемблирования более подробно описаны в руководстве по макроассемблеру MASM 6.14 фирмы Microsoft.

# 2.2. Компоновка программ

После успешной трансляции исходного текста ассемблерной программы результат в виде объектного файла передается компоновщику link.exe. Компоновщик может связать несколько объектных файлов в один исполняемый EXE-файл. При этом все сегменты, определенные в программе, группируются в соответствии с инструкциями, содержащимися в объектном файле. Вся информация о размещении сегментов записывается в заголовок исполняемого файла.

Здесь следует упомянуть о том, что структура программы (не только на ассемблере) определяется несколькими факторами:

- архитектурой процессора;
- особенностями той операционной системы, под управлением которой эта программа будет выполняться;
- правилами работы выбранного компилятора — разные компиляторы предъявляют разные требования к исходному тексту программы.

Например, исходный текст простой 16-разрядной программы, выводящей строку str на экран в операционной системе MS-DOS, может выглядеть так:

```
assume CS:code. DS:data
code  segment
start:
 mov  AX. data
 mov  DS. AX
 mov  AH. 09h
 mov  DX. offset str
 int  21h
 mov  AX. 4C00h
 int  21h
code  ends
data  segment
  str DB "Test string$"
data  ends
end   start
```

Эта программа выполняется только в операционной системе MS-DOS и не работает в таких операционных системах, как Windows 2000 и Windows XP, поскольку структура исполняемого файла не соответствует требованиям, выдвигаемым этими операционными системами. Чтобы программа могла вывести строку на экран, например, в Windows XP, требуется кардинальным образом изменить структуру программы.

Кроме этого, программы для операционных систем MS-DOS и Windows требуют задания различных параметров компилятора и компоновщика, что вызвано различной организацией операционных систем MS-DOS и Windows. Операционная система MS-DOS использует 16-разрядную модель памяти в реальном режиме, в то время как Windows XP, например, 32-разрядный защищенный режим с линейной адресацией памяти. Далее мы проанализируем основные параметры компилятора **ml.exe** и компоновщика **link.exe** макроассемблера MASM для создания различных типов приложений.

Трансляцию файлов с расширением **ASM** можно выполнить из командной строки:

```
ml /c /coff имя_файла.asm
```

Созданный при помощи этой команды объектный файл имеет формат COFF. Если параметр /coff не задан, то форматом созданного объектного файла будет OMF.

Компоновщик **link.exe** оперирует с OBJ-файлами как в формате COFF, так и в формате OMF, при этом выполняется автоматическое преобразование формата файла из OMF в COFF. Обычно при генерации исполняемых файлов используется формат COFF. Кроме того, и это очень важно, если файл объектного модуля должен применяться в приложении, написанном на Visual C++ .NET, то формат его обязательно должен быть COFF. В то же время при применении объектного файла в приложении, разработанном в Borland Delphi 2005, единственным воспринимаемым форматом является OMF.

Для того чтобы из объектного файла создать исполняемый файл, работающий в MS-DOS, следует выполнить командную строку

```
link /co имя_файла.obj
```

Здесь следует учитывать то, что версия компоновщика **link.exe** должна поддерживать генерацию 16-разрядных приложений MS-DOS. Использование 32-разрядных компоновщиков приведет к ошибке создания EXE-файла.

Для генерирования 32-разрядных EXE-файлов следует использовать корректную версию компоновщика, при этом командная строка должна выглядеть так:

```
link /SUBSYSTEM:WINDOWS /OPT:NOREF имя_файла.obj
```

Приведенных здесь сведений вполне достаточно для компиляции исходных текстов ассемблерных программ и процедур, представленных в книге. Более подробная информация о пакете MASM доступна в Интернете, а также в многочисленных литературных источниках.

В последующих главах мы рассмотрим структуру данных и синтаксис команд макроассемблера MASM.



# Синтаксис языка ассемблера




Язык ассемблера, называемый также языком символического кодирования, представляет собой машинный язык в символической форме, которая более понятна и удобна программисту. Сложная внутренняя структура, разнообразные форматы команд, многочисленные режимы адресации процессоров Intel ограничивают возможности разработки сложных и объемных программ на языке ассемблера. Однако в любых сложных программах всегда существуют критические секции, требующие интенсивных и быстрых вычислений, которые приходится разрабатывать не на языке высокого уровня, а на ассемблере.

Разработка программ на языке ассемблера требует хороших знаний архитектуры всей системы, включая режимы адресации данных, структуры памяти и системы команд процессора. Поэтому анализ возможностей ассемблера мы будем проводить в тесной взаимосвязи с архитектурой процессоров Intel Pentium. Напомню, что мы рассматриваем язык ассемблера, основанный на разработке фирмы Microsoft версии 6.14 и выше. Он включает в себя множество параметров, команд и директив, анализ которых займет много времени. Поэтому здесь будут рассмотрены только наиболее важные языковые конструкции, без знания которых создавать программы невозможно. Для такого анализа нужно четко понимать, как данные представляются в компьютере и каковы общие принципы их обработки, поэтому начнем именно с этого.

## 3.1. Представление данных в компьютере

В основе работы компьютера лежат понятия бита и байта — именно они представляют данные и команды в памяти. Минимальной единицей информации в компьютере является бит. Бит может принимать одно из двух значений: 0 или 1 — и является составным элементом для более информативных единиц данных. Минимальный объем информации, к которому имеется доступ в памяти компьютера, составляет

один байт (8 двоичных разрядов, или битов), при этом говорят о байтовой организации памяти (хотя теоретически память может быть организована и по-другому). Все байты оперативной памяти нумеруются начиная с нуля. Местоположение каждого байта в памяти характеризуется его номером или, по-другому, адресом. Схематически байт представляет собой структуру данных, изображенную на рис. 3.1.

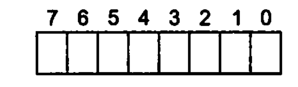



**Рис. 3.1.** Представление байта

Старший бит байта имеет номер 7, младший — 0. В оперативной памяти машины байты данных располагаются по возрастанию адресов (рис. 3.2).

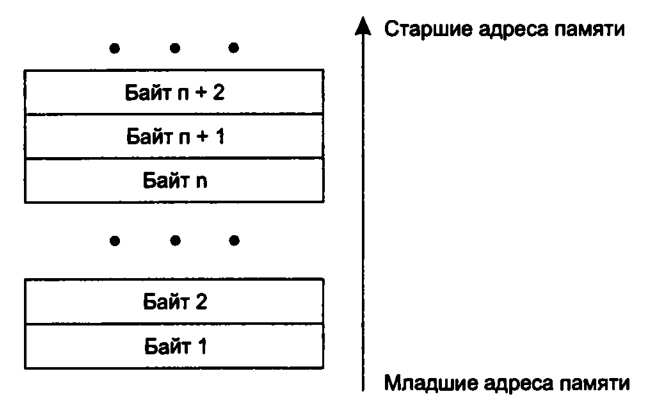



**Рис. 3.2.** Адресация памяти

Исключение составляет специальная область памяти, называемая стеком, — в ней байты данных располагаются в сторону убывания адресов. Более подробно мы рассмотрим стек в последующих главах.

Редко случается так, что для представления данных требуется один байт. Во многих случаях данные нужно представить бóльшим числом байтов. Если данные можно представить двумя байтами, то говорят, что они представлены словом. О данных, требующих для представления 4 байта, говорят, что они имеют размерность двойного слова. Наконец, данные могут быть представлены восемью (учетверенное слово) или шестнадцатью (двойное учетверенное слово) байтами. Во всех этих случаях расположение байтов соответствует правилу: младший байт располагается по младшему адресу, а старший байт — по старшему (рис. 3.3).

Нумерация байтов в обычных, двойных и учетверенных словах начинается с младшего (нулевого) байта и заканчивается 1, 3 и 7-м байтом соответственно. В документации часто используется такой способ расположения байтов, когда старшие байты располагаются слева, а младшие — справа. Пример такого расположения байтов в двойном слове показан на рис. 3.4.

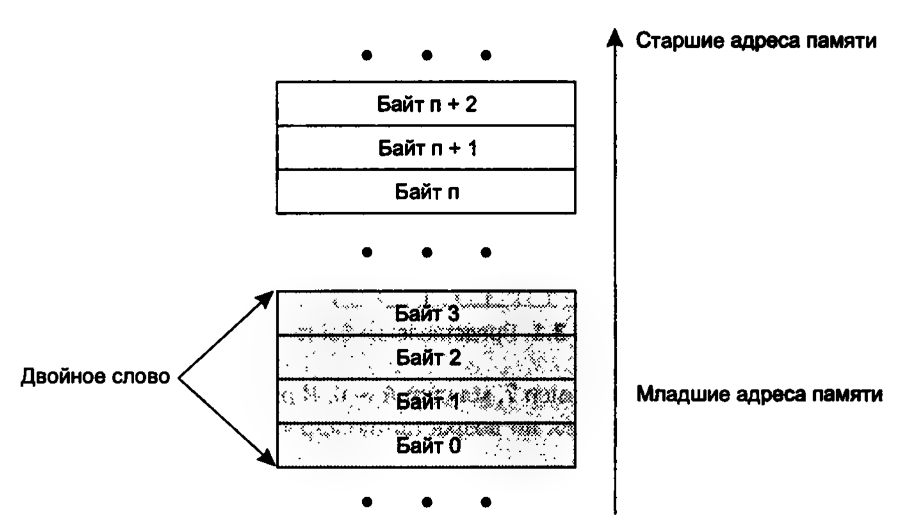



**Рис. 3.3.** Представление данных различной размерности в памяти

Крайний слева байт принято называть старшим, а крайний справа — младшим. Такой порядок расположения байтов связан с обычной для нас формой записи чисел, когда в многоразрядном числе слева находятся старшие разряды, а справа — младшие. Следующее число опять начнется со старшего разряда и закончится младшим. Однако в памяти компьютера данные располагаются в более естественном порядке непрерывного возрастания номеров байтов, и, таким образом, каждое слово или двойное слово в памяти начинается с младшего байта и заканчивается старшим (см. рис. 3.4).

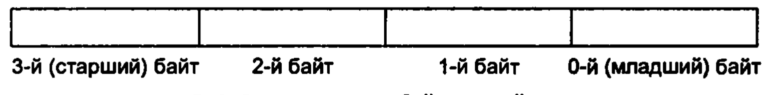



**Рис. 3.4.** Расположение байтов двойного слова

Вкратце напомню, как интерпретируются числовые данные в компьютере. Комбинируя двоичные цифры (биты), можно представить любое числовое значение. Значение двоичного числа определяется относительной позицией каждого бита и наличием единичных битов. Рассмотрим восьмибитовое число (байт), представленное следующим образом:

10100101

Поскольку мы имеем дело с двоичной системой счисления, то это число можно представить так:

$$1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0.$$

Значение этого числа в десятичной системе равно 165. Таким образом, любое двоичное число, имеющее *n* разрядов, можно представить в виде

$$k \times 2^{n-1} + k \times 2^{n-2} + \ldots + k \times 2^0.$$

Здесь $k$ может принимать одно из двух значений: 0 или 1. Разрядность $n$ двоичного числа определяется архитектурой системы и обычно кратна восьми. Сразу замечу, что мы рассматриваем двоичное представление целых чисел, являющееся базисом для понимания вычислительных операций с любыми другими типами чисел, такими, например, как вещественные числа или, в терминологии ассемблера, «числа с плавающей точкой».

В арифметических операциях задействованы положительные и отрицательные целые и вещественные числа, поэтому необходимо каким-то образом различать их знаки. Знак двоичного числа указывается старшим или, как его называют, знаковым битом числа. Положительные числа имеют в старшем разряде нулевой бит, а отрицательные числа — единичный. Отрицательные двоичные числа выражаются двоичным дополнением, то есть для представления отрицательного двоичного числа необходимо инвертировать все его биты и к результату прибавить 1.

В следующем примере находится двоичное представление числа −61. Положительное число 61 представляется как 00111101, а процесс преобразования показан далее:

```
11000010 (инверсия числа 61)
+
00000001
-----------
11000011 (-61)
```

Несколько слов об операции сложения. Она выполняется по простым правилам:

0 + 0 = 0
1 + 0 = 1
0 + 1 = 1
1 + 1 = 0 + 1 (бит переноса)

Как и в десятичной системе счисления, при выходе за пределы разрядной сетки для данного разряда образуется единица переноса в следующий разряд. Это продемонстрировано на рис. 3.5.

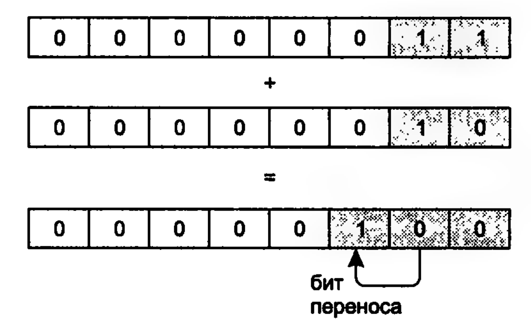



**Рис. 3.5.** Схема сложения двоичных чисел с переносом

Проверить результат преобразования положительного числа в отрицательное очень просто: достаточно сложить оба числа, при этом результат должен быть нулевым. Например, если сложить числа 61 и –61, должен получиться 0:

00111101 (61)

+

11000011 (–61)

------------

00000000

Результат получился нулевым, что свидетельствует о корректности преобразования. Перенос из самого старшего разряда при этом теряется.

Вычитание двоичных чисел выполняется как модифицированный вариант сложения, при этом вначале инвертируется знак вычитаемого, после чего числа складываются. Это обусловлено тем, что операционный блок процессора содержит только устройства сложения (сумматоры) и не имеет устройств вычитания.

Приведу простой пример. Пусть требуется из числа 5 вычесть 2. Эту операцию можно представить как 5 + (–2). Число 5 представляется в двоичной форме как 00000101, а число –2 — как 11111110. Результат вычисляется следующим образом:

00000101 (5)

+

11111110 (–2)

-----------

00000011 (3)

Здесь я хочу сделать важное замечание. Процессор ничего не «знает» о знаковых и беззнаковых числах, он просто складывает биты операндов, поэтому вся ответственность за интерпретацию результатов ложится на прикладные программы. Операции умножения и деления алгоритмически более сложны, но в их основе также лежат операции сложения и вычитания.

Представление двоичных чисел в виде последовательности нулей и единиц часто бывает не очень удобным из-за своей громоздкости и не очень хорошей читабельности. Во многих случаях используется так называемое шестнадцатеричное представление чисел. Такая система счисления включает символы от 0 до F и, поскольку таких символов 16, называется шестнадцатеричной. Шестнадцатеричный формат нашел широкое применение в языке ассемблера. В ассемблерных листингах программ в шестнадцатеричном формате показаны все адреса, машинные коды команд и содержимое констант. Отладочная информация также выдается в шестнадцатеричном формате.

В табл. 3.1 приведены десятичные, двоичные и шестнадцатеричные значения чисел от 0 до 15.

Если немного поработать с шестнадцатеричным форматом, то можно быстро к нему привыкнуть.

**Таблица 3.1.** Соответствие между десятичными, двоичными и шестнадцатеричными числами

| Десятичное число | Двоичное число | Шестнадцатеричное число |
|---|---|---|
| 0 | 0000 | 0 |
| 1 | 0001 | 1 |
| 2 | 0010 | 2 |
| 3 | 0011 | 3 |
| 4 | 0100 | 4 |
| 5 | 0101 | 5 |
| 6 | 0110 | 6 |
| 7 | 0111 | 7 |
| 8 | 1000 | 8 |
| 9 | 1001 | 9 |
| 10 | 1010 | A (a) |
| 11 | 1011 | B (b) |
| 12 | 1100 | C (c) |
| 13 | 1101 | D (d) |
| 14 | 1110 | E (e) |
| 15 | 1111 | F (f) |

Для того чтобы различать форматы чисел, в языке ассемблера приняты специальные обозначения: B, b — двоичные числа; H, h — шестнадцатеричные числа.

Приведу несколько примеров чисел в разных форматах:

56 = 00111000b = 38h
−13 = 11110101 = F5h

Сложение и вычитание чисел в шестнадцатеричном формате осуществляется по тем же правилам, что и двоичных или десятичных чисел: операция выполняется для каждого разряда с учетом переноса из младшего разряда или заема из старшего. Рассмотрим несколько примеров.

Пусть требуется сложить два числа в шестнадцатеричном формате: 3Fh и 27h:

```
3F
+
27
---
66
```

При сложении младших разрядов, равных F и 7, результирующее значение равно 22 (в десятичной системе), то есть младший разряд будет равен 22 – 16 = 6, при этом происходит перенос в старший разряд. При сложении старших разрядов результирующее значение вычисляется как 3 + 2 + бит переноса, то есть окончательный результат равен 66h.

В следующем примере необходимо вычесть шестнадцатеричное значение 7Eh из AAh:

AA
–
7E
---
2C

При вычитании младших разрядов, равных A (10 в десятичной системе) и E (14 в десятичной системе), необходим заем из старших разрядов. Тогда значение младшего разряда будет равно 16 + 10 – 14 = 12 или в шестнадцатеричной форме — C. Результат вычитания старших разрядов будет равен 9 – 7 = 2. Окончательный результат вычитания равен 2Ch.

Двоичные числа используются не только в вычислениях, но и для другой функции — с их помощью можно выводить информацию в символьном представлении на экран дисплея или периферийное устройство печати. Для стандартного представления таких символов используется код ASCII (American National Standard Code for Information Interchange — Американский национальный стандартный код для обмена информацией).

Представление символа A в соответствии со стандартом ASCII выражается шестнадцатеричным значением 41h, представление символа B — значением 42h и т. д. Наличие стандартного кода облегчает обмен данными между различными устройствами компьютера. При этом 8-битовый расширенный код ASCII, используемый в компьютерах, обеспечивает представление 256 символов, включая символы национальных алфавитов.

## 3.2. Первичные элементы языка ассемблера

Все ассемблерные программы состоят из одного или более предложений и комментариев. Предложение и комментарий представляют собой комбинацию знаков, входящих в алфавит языка, а также чисел и идентификаторов, которые тоже формируются из знаков алфавита. Макроассемблер MASM распознает следующий набор знаков:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
? @ _ $ : . [ ] ( ) < > { }
+ / * & % ! ' ~ | \ = # ^ ; . ` "
```

Конструкции языка ассемблера формируются из идентификаторов и ограничителей. Идентификатор представляет собой набор букв, цифр и символов _, ?, $ или @, причем первый элемент не должен быть цифрой. Идентификатор должен полностью размещаться на одной строке и может содержать от 1 до 31 символа (если их больше чем 31, то остальные игнорируются).

Друг от друга идентификаторы отделяются пробелом или ограничителем, которым считается любой недопустимый в идентификаторе символ. Посредством

идентификаторов представляются объекты программы, такие, как переменные, метки и имена.

Переменные идентифицируют находящиеся в памяти данные и в общем случае характеризуются тремя атрибутами:

- сегментом, в котором определена переменная (действительно для 16-разрядных приложений, где полный адрес переменной формируется как *сегмент: смещение*; для 32-разрядных приложений этот атрибут не используется);
- смещением данного поля памяти относительно начала сегмента;
- типом, определяющим число, обрабатываемое при работе с переменной.

Метка (label) является частным случаем переменной, причем ссылка на нее указывается в командах условного или безусловного перехода. Для 16-разрядных приложений метка характеризуется атрибутами *сегмент:смещение*, для 32-разрядных — только смещением. Метка также может быть определена через другую метку с использованием директивы EQU, как в этом примере:

```
var1 EQU   label1
. . .
label1:
    mov  AX, 1
. . .
```

Именами являются последовательности символов, определенные директивой EQU и принимающие значение символа или числа. Другое название имени — константа. Примеры имен:

```
name1  EQU 'ABCD'
digit  EQU 10
```

Некоторые идентификаторы, называемые ключевыми словами, имеют предопределенный смысл. К ним относятся директивы ассемблера, команды (инструкции) процессора, имена регистров, операторы выражений. К таким идентификаторам относится и указатель позиции (location counter), обозначаемый символом $.

Указатель позиции представляет собой текущую позицию в текущем сегменте и имеет те же атрибуты, что и метка типа NEAR. Далее приведен пример использования указателя позиции:

```
string1 BYTE   "Test String"
level   WORD   5
res     BYTE   10 DUP (?)
len     EQU    $-string1
```

Константа len в этом примере равна 22 (именно столько байтов памяти занимают переменные string1, level и res).

Следует помнить, что обычно ассемблер не различает строчные и прописные буквы и идентификаторы могут включать в себя буквы обоих регистров. Например, идентификаторы Abs и abs считаются идентичными. Различие между строчными и прописными буквами может быть установлено параметрами /ML и /MX макроассемблера MASM.

Рассмотрим типы и формы представления данных, которые могут быть использованы в выражениях, директивах и инструкциях языка ассемблера. Начнем

с целых чисел. Целые числа могут быть представлены набором цифр и/или символов, после которых задается тип кодировки (основание счисления). Тип кодировки определяется одной из литер: B — двоичная, O — восьмеричная, D или T — десятичная, H — шестнадцатеричная. При этом шестнадцатеричные числа не должны начинаться с буквенных шестнадцатеричных цифр (например, вместо ABh следует использовать запись 0ABh). Шестнадцатеричные цифры от A до F могут кодироваться в обоих регистрах.

Процессор оперирует с типами данных, определяемых директивами:

- DB — распределение и инициализация 1 байта памяти для каждого из указанных значений. В качестве значения может кодироваться целое число, строковая константа, оператор DUP (см. далее), абсолютное выражение или знак ?. Знак ? обозначает неопределенное значение. Значения, если их несколько, должны разделяться запятыми. Если директива имеет имя, создается переменная типа BYTE с соответствующим данному значению указателя позиции смещением. Строковая константа может содержать столько символов, сколько помещается на одной строке. Символы строки хранятся в памяти в порядке их следования, то есть первый символ имеет самый младший адрес, последний — самый старший;
- DW — распределение и инициализация слова памяти (2 байта) для каждого из указанных значений. В качестве значения может кодироваться целое число, одно- или двухсимвольная константа, оператор DUP, абсолютное выражение, адресное выражение или знак ?. Знак ? обозначает неопределенное значение. Значения, если их несколько, должны разделяться запятыми. Если директива имеет имя, создается переменная типа WORD с соответствующим данному значению указателя позиции смещением. Строковая константа не может содержать более двух символов. Последний (или единственный) символ строки хранится в младшем байте слова. Старший байт содержит первый символ или, если строка односимвольная, ноль;
- DD — распределение и инициализация двойного слова памяти (4 байта) для каждого из указанных значений. В качестве значения может кодироваться целое число, одно- или двухсимвольная константа, действительное число, кодированное действительное число, оператор DUP, абсолютное выражение, адресное выражение или знак ?. Знак ? обозначает неопределенное значение. Значения, если их несколько, должны разделяться запятыми. Если директива имеет имя, создается переменная типа DWORD с соответствующим данному значению указателя позиции смещением. Строковая константа не может содержать более двух символов. Последний (или единственный) символ строки хранится в младшем байте слова. Второй байт содержит первый символ или, если строка односимвольная, ноль. Остальные байты заполняются нулями;
- DQ — распределение и инициализация 8 байт памяти для каждого из указанных значений. В качестве значения может кодироваться целое число, одно- или двухсимвольная константа, действительное число, кодированное действительное число, оператор DUP, абсолютное выражение, адресное выражение или знак ?. Знак ? обозначает неопределенное значение. Значения, если их

несколько, должны разделяться запятыми. Если директива имеет имя, создается переменная типа QWORD с соответствующим данному значению указателя позиции смещением. Строковая константа не может содержать более двух символов. Последний (или единственный) символ строки хранится в младшем байте слова. Второй байт содержит первый символ или, если строка односимвольная, ноль. Остальные байты заполняются нулями;

- DT — распределение и инициализация 10 байт памяти для каждого из указанных значений. В качестве значения может кодироваться целое выражение, упакованное десятичное число, одно- или двухсимвольная константа, кодированное действительное число, оператор DUP или знак ?. Знак ? обозначает неопределенное значение. Значения, если их несколько, должны разделяться запятыми. Если директива имеет имя, создается переменная типа TWORD с соответствующим данному значению указателя позиции смещением. Строковая константа не может содержать более двух символов. Последний (или единственный) символ строки хранится в младшем байте слова. Второй байт содержит первый символ или, если строка односимвольная, ноль. Остальные байты заполняются нулями. При обработке директивы DT подразумевается, что константы, содержащие десятичные цифры, представляют собой не целые, а десятичные упакованные числа. Чтобы в случае необходимости определить 10-байтовое целое число, следует после числа указать спецификатор системы счисления (D или d для десятичных чисел, H или h для шестнадцатеричных и так далее). Если в одной директиве определения памяти заданы несколько значений, им распределяются последовательные байты памяти.

Во всех директивах определения памяти в качестве одного из значений может быть задан оператор DUP. Он имеет следующий формат:

```
счетчик DUP (значение, ...)
```

Указанный в скобках список *значений* повторяется многократно в соответствии со значением *счетчика*. Каждое значение в скобках может быть любым выражением, например целым числом, символьной константой или другим оператором DUP (допускается до 17 уровней вложенности операторов DUP). Значения, если их несколько, должны разделяться запятыми. Оператор DUP может использоваться не только в директивах определения памяти, но и в других директивах.

Далее приводятся примеры директив определения данных:

```
db1  DB 1
db2  DB 'ABCD'
db3  DB ?
dw1  DW 9325
dw2  DW 4*3
dw3  DW 1,'$'
dw4  DW array
dd1  DD 'xyz'
dd2  DD 1.5
dq1  DQ 18446744073709551615
mix1 DB 5 DUP(5 DUP(5 DUP(10)))
mix2 DW DUP(1,2,3,4,5)
```

Рассмотрим способы представления (кодировки) для различных типов данных. Вещественные числа (числа с плавающей точкой) кодируются с использованием одной из форм:

```
xxxx.xxxx[R]
[[+|-]]xxxx.xxxx[[E[[+|-]]xxxx]]
```

Здесь *x* — одна из цифр от 0 до 9.

Кодированные вещественные числа могут использоваться в директивах DD, DQ и DT, например:

```
a1  DD 56.23R
a2  DD -45.6R
a3  DD 211.77E-2
```

Десятичные числа кодируются так, как показано далее:

```
[[+|-]]xxxx
```

Здесь *x* — одна из цифр от 0 до 9.

Следующий тип данных, который мы рассмотрим, — знаковые и строковые константы.

Они могут быть представлены следующим образом:

```
'cccccc'
"ccccc"
```

Здесь *c* — символы из допустимого диапазона.

Часть строки исходного текста после символа точки с запятой (если он не является элементом знаковой константы или строки знаков) считается комментарием и ассемблером игнорируется. Комментарии вносятся в программу как поясняющий текст и могут содержать любые знаки до ближайшей комбинации символов возврата каретки и перевода строки (CR/LF).

В языке ассемблера очень часто используются так называемые символические имена, которые существенно упрощают программирование. Описание символических имен выполняется с помощью специальных директив.

Символические имена могут представлять собой число, текст, инструкцию или адрес. Для описания символических имен в языке ассемблера служат директивы EQU, LABEL и директива присваивания =.

Директива присваивания имеет такой формат:

```
имя=выражение
```

При помощи этой директивы создается *имя*, представляющее собой *значение*, равное текущему значению указанного выражения. Для хранения этого значения память не выделяется. Вместо этого каждое появление указанного имени в программном коде замещается значением выражения.

Символическое имя может быть переопределено. В каждой директиве присваивания в качестве имени может указываться уникальное имя или имя, ранее использованное другой директивой присваивания.

Выражение может быть целым числом, одно- или двухсимвольной строковой константой, константным или адресным выражением. Его значение не должно превышать 65 535.

Примеры определения символических имен:

```
int = 167
string1 = 'ab'
const = 7*4
addr1  = string1
```

Директива EQU имеет следующий формат:

*имя* EQU *выражение*

Директива EQU создает абсолютное имя, псевдоним или текстовое имя путем присваивания *имени* указанного *выражения*. Под абсолютным именем здесь подразумевается имя, представляющее собой 16-разрядное значение, а псевдонимом называется ссылка на другое имя. В качестве текстового имени может использоваться строка символов. При компиляции исходного текста каждое появление имени ассемблер замещает текстом или значением выражения, в зависимости от типа выражения. Имя должно быть уникальным и не может быть переопределено. В качестве выражения может задаваться целое число, строковая константа, вещественное число, кодированное вещественное число, мнемоника инструкции, константное или адресное выражение. Выражение, имеющее значением целое число, порождает имя, вхождения которого ассемблер замещает этим значением. Для всех остальных выражений вхождения имени замещаются текстом.

Примеры применения директивы EQU:

```
k     EQU 1024
adr   EQU [BP]
cle   EQU XOR AX.AX
d     EQU BYTE PTR
t     EQU 'File'
MASM  EQU 5.1 + 0.9
Msft  EQU <Microsoft>
mat   EQU 20*30
```

Директива LABEL имеет следующий формат:

*имя* LABEL *тип*

Директива LABEL порождает новую переменную или метку путем присваивания *имени* указанного *типа* и текущего значения указателя позиции. Имя должно быть уникальным и не может быть переопределено. В качестве типа может быть задано одно из следующих ключевых слов: BYTE, WORD, DWORD, QWORD, TBYTE, NEAR, FAR.

Примеры использования директивы LABEL:

```
byte_array   LABEL BYTE
word_array DW     10 DUP(0)
```

Здесь имена byte_array и word_array ссылаются на одну и ту же область памяти.

# 3.3. Программная модель процессора Intel Pentium

Для понимания работы команд ассемблера необходимо четко представлять, как выполняется адресация данных, какие регистры процессора и как могут использоваться при выполнении инструкций. Сейчас мы рассмотрим базовую программную модель процессоров Intel Pentium, в которую входят:

- 8 регистров общего назначения, служащих для хранения данных и указателей;
- регистры сегментов — они хранят 6 селекторов сегментов;
- регистр управления и контроля EFLAGS, который позволяет управлять состоянием выполнения программы и состоянием (на уровне приложения) процессора;
- регистр-указатель EIP выполняемой следующей инструкции процессора;
- система команд (инструкций) процессора;
- режимы адресации данных в командах процессора.

Начнем с описания базовых регистров процессора Intel Pentium.

Базовые регистры процессора Intel Pentium являются основой для разработки программ и позволяют решать основные задачи по обработке данных. Все они показаны на рис. 3.6.




**Рис. 3.6.** Базовые регистры процессора Intel Pentium

Среди базового набора регистров выделим отдельные группы и рассмотрим их назначение. Начнем с 32-разрядных регистров общего назначения (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP), которые могут использоваться в качестве:

- операндов в арифметических и логических операциях;
- операндов при вычислении адресов операндов;
- указателей на переменные в памяти.

Несмотря на то что любой из этих регистров можно использовать для вышеперечисленных операций, необходимо учитывать специфику применения регистра ESP — он служит для хранения указателя стека, поэтому задействовать его для других целей не рекомендуется. Во многих программах требуется хранить в регистрах определенные значения операндов в процессе выполнения каких-либо операций. Для этого подходят регистры ECX, ESI и EDI. Если при этом используется отдельный сегмент данных, то некоторые команды ассемблера предполагают, что переменная, адресуемая одним из этих регистров, находится в сегменте данных, определяемом регистром DS.

Во многих случаях регистры общего назначения используются для предопределенных целей:

- EAX выполняет функцию аккумулятора при работе с операндами и хранит результат операции;
- EBX — указатель на данные, находящиеся в сегменте данных, адресуемом регистром DS;
- ECX — счетчик циклов и элементов при строковых операциях;
- EDX — указатель на порты устройств ввода-вывода;
- ESI — указатель на данные, находящиеся в сегменте, адресуемом регистром DS (при выполнении строковых операций содержит смещение строки-источника);
- EDI — указатель на данные, находящиеся в сегменте, адресуемом регистром ES (при выполнении строковых операций содержит смещение строки-приемника);
- ESP содержит указатель стека в сегменте стека, адресуемом регистром SS;
- EBP содержит указатель на данные, находящиеся в стеке, адресуемом регистром SS.

Младшие 16 бит 32-разрядных регистров общего назначения могут адресоваться так же, как и 16-разрядные регистры в процессорах 8086 с именами AX, BX, CX, DX, BP, SI, DI, SP. В свою очередь, 16-разрядные регистры AX, BX, CX и DX позволяют обращаться отдельно как к старшим 8-разрядным регистрам (AH, BH, CH, DH), так и к младшим (AL, BL, CL, DL). Это проиллюстрировано рис. 3.7.

Сегментные регистры (CS, DS, SS, ES, FS и GS) содержат 16-разрядные селекторы сегментов. Селектор представляет собой специальный указатель, который

идентифицирует данный сегмент в памяти. Более подробно мы остановимся на использовании сегментных регистров в главе 4.

| 31 — 16 | 15 — 8 | 7 — 0 | 31 бит | 16 бит | 8 бит |
|---|---|---|---|---|---|
| | AH | AL | EAX | AX | AH/AL |
| | BH | BL | EBX | BX | BH/BL |
| | CH | CL | ECX | CX | CH/CL |
| | DH | DL | EDX | DX | DH/DL |
| | SI | | ESI | | |
| | DI | | EDI | | |
| | BP | | EBP | | |
| | SP | | ESP | | |

**Рис. 3.7.** Использование регистров общего назначения

Помимо регистров общего назначения и сегментных регистров, как мы знаем, в базовой архитектуре имеется еще два регистра: регистр управления/состояния EFLAGS и регистр-указатель адреса следующей инструкции EIP. Рассмотрим их более подробно.

Регистр EFLAFS, часто именуемый регистром флагов, имеет 32 разряда и содержит группу битов или, как их чаще называют, флагов состояния, управляющий флаг и группу системных флагов. На рис. 3.8 показана структура регистра флагов и приводится расшифровка наиболее часто используемых флагов.

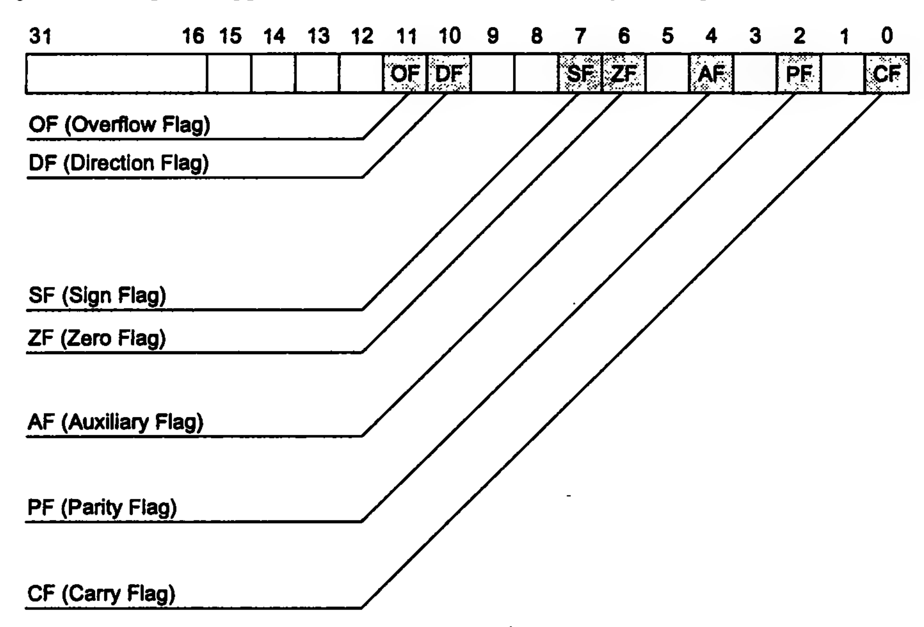



**Рис. 3.8.** Регистр управления/состояния процессора

Указанные на рис. 3.8 флаги наиболее часто используются в прикладных программах и сигнализируют о следующих событиях:

- OF (флаг переполнения) фиксирует ситуацию переполнения, то есть выход результата арифметической операции за пределы допустимого диапазона значений;
- DF (флаг направления) используется командами обработки строк. Если DF = 0, строка обрабатывается в прямом направлении, от меньших адресов к большим. Если DF = 1, обработка строк ведется в обратном направлении;
- SF (флаг знака) показывает знак результата операции, при отрицательном результате SF = 1;
- ZF (флаг нуля) устанавливается в 1, если результат операции равен 0;
- AF (флаг вспомогательного переноса) используется в операциях над упакованными двоично-десятичными числами. Этот флаг служит индикатором переноса или заема из старшей тетрады (бит 3);
- PF (флаг четности) устанавливается в 1, если результат операции содержит четное число двоичных единиц;
- CF (флаг переноса) показывает, был ли перенос или заем при выполнении арифметических операций.

Из всех этих флагов только флаг переноса CF может устанавливаться или сбрасываться непосредственно при помощи команд ассемблера stc, clc и cmc. Кроме того, в этот флаг может быть скопирован бит, определенный командами bt, bts, btr и btc.

Флаги состояния также используются при анализе операций, результатами которых являются беззнаковые целые числа, целые числа со знаком и упакованные (BCD) целые числа. Если результатом операции является беззнаковое целое число, то установка флага переноса CF в 1 (перенос или заем) свидетельствует о выходе за пределы допустимого диапазона. Если результатом операции является целое число со знаком (двоичное дополнение числа), то об этом свидетельствует установка в 1 флага переполнения OF.

В случае если результат операции интерпретируется как число в формате BCD, то установка флага AF свидетельствует о возникновении переноса или заема. Флаг SF указывает на знак результата, являющегося знаковым числом. Флаг ZF указывает на равенство нулю знакового или беззнакового числа.

При выполнении операций целочисленной арифметики с повышенной точностью флаг переноса CF используется командами adc (сложение с переносом) и sbb (вычитание с заемом) для того, чтобы учитывать перенос при переходе к следующей операции сложения или вычитания.

Флаги состояния используются командами условного перехода j*CC* (*CC* — код условия: eq, le, lt, ne и т. д.), командами set*CC*, loop*CC* и cmov*CC*.

Флаги состояния процессора могут быть помещены в стек и извлечены из стека командами pushf, pushfd, popf, popfd. Кроме того, флаги могут быть загружены в старшую половину регистра AX или извлечены из старшей половины при помощи команды lahf или sahf.

Перейдем к описанию регистра EIP — он содержит смещение в программном сегменте следующей выполняемой команды. Если в программе встречаются команды j*CC*, call, ret или iret, то содержимое регистра EIP может измениться произвольным образом — смещение следующей команды может быть как положительным, так и отрицательным. Содержимое регистра-указателя следующей команды не может быть изменено какой-либо инструкцией напрямую, хотя можно получить его содержимое, если выполнить команду call, а затем прочитать указатель на следующую команду, находящийся в стеке.

Регистр EIP можно модифицировать, опять-таки не прямо, а через стек, заменив адрес следующей команды. Естественно, перед этим следует выполнить команду call.

Прежде чем приступить к анализу команд ассемблера и способов обработки данных, нам необходимо рассмотреть модели памяти, с которыми может работать процессор. Модель памяти определяет способ организации программ и данных в памяти компьютера. В 32-разрядной архитектуре процессора Intel Pentium используются три модели памяти:

- плоская, или линейная, модель памяти (flat memory model) — память представляет собой непрерывное пространство адресов. Такое пространство адресов называется линейным. Программный код, данные, область стека располагаются в этом пространстве адресов. Адресное пространство в этой модели адресуется побайтно, а диапазон адресов равен $2^{32}$. Схематично эта модель памяти показана на рис. 3.9;

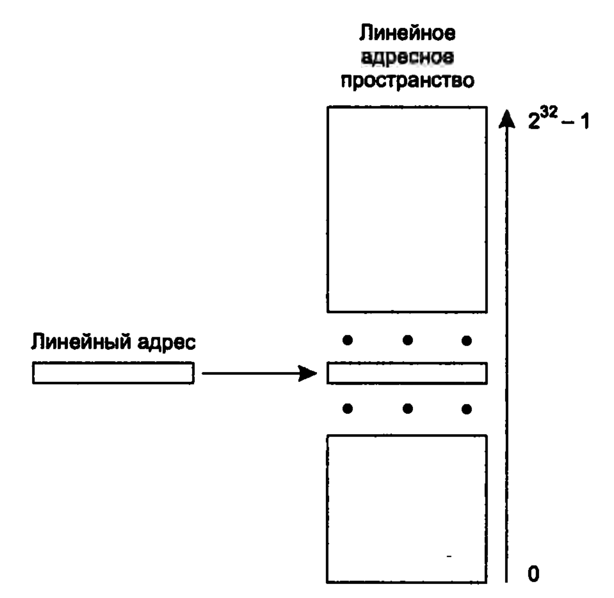



**Рис. 3.9.** Линейная модель памяти

- сегментированная модель памяти (segmented memory model) — память состоит из трех отдельных пространств адресов, которые называются сег-

ментами. При этом программный код, данные и стек размещаются в отдельных сегментах памяти. Для того чтобы обратиться к байту в памяти, программа формирует логический адрес, состоящий из адреса сегмента (селектора сегмента) и смещения. Программы, выполняющиеся в 32-разрядном режиме, могут использовать до 16 383 сегментов разного размера, каждый из которых может иметь размер $2^{32}$ байт. Схема адресации для сегментированной модели памяти показана на рис. 3.10.

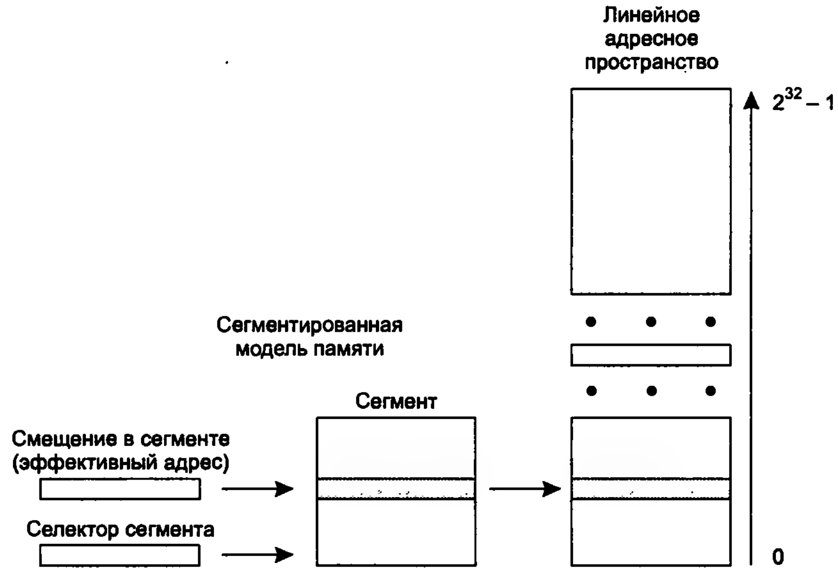



**Рис. 3.10.** Сегментированная модель памяти

Механизм преобразования адресов при использовании сегментированной модели таков: вначале все адреса сегментов, определенных в программе, отображаются на общее линейное пространство адресов, доступное процессору. Затем логический адрес операнда в памяти преобразуется в линейный адрес из пространства адресов. Все эти преобразования прозрачны для работающей программы. Подобная модель позволяет повысить надежность работы программ. Например, если разместить программный код и область стека в разных сегментах, то в случае резкого увеличения размера стека не произойдет перекрытия программного кода данными из стека;

- модель реального режима адресации (real-address mode memory model) — это модель памяти, используемая в процессорах 8086. Данная модель памяти поддерживается для того, чтобы обеспечить совместимость с ранее разработанными 16-разрядными приложениями. В этой модели применяется механизм сегментации, причем максимальный размер сегмента не превышает 64 Кбайт. Максимальный размер линейного адресного пространства, доступного в этом режиме, равен $2^{20}$ байт.

При разработке 32-разрядных программ на языке ассемблера обычно используется линейная, или плоская, модель памяти. Все примеры 32-разрядных процедур, приведенные в этой и последующих главах, разработаны с использованием этой модели. При 32-разрядной адресации операндов логический адрес состоит из 16-разрядного селектора сегмента и 32-разрядного смещения. При 16-разрядной адресации логический адрес состоит из 16-разрядного селектора сегмента и 16-разрядного смещения.

При выполнении любой программы процессор обращается к памяти, в которой хранятся команды и данные. Для доступа к данным необходимо каким-то образом определять их адрес в памяти. Способ формирования адреса операнда или метки перехода на другую команду называется режимом адресации, или адресацией.

Инструкции ассемблера включают самые разнообразные команды, работающие как с операндами, так и без них. Некоторые команды требуют явного указания операндов, в то время как другие используют операнды по умолчанию. Данные операнда-источника могут находиться в регистре, памяти, в порту ввода-вывода или задаваться непосредственно в инструкции. Операнд-приемник может располагаться в оперативной памяти, в регистре или быть портом ввода-вывода.

Для четкого понимания того, как осуществляется адресация данных, проанализируем способ образования адреса операнда. Адрес операнда формируется по схеме *сегмент:смещение*. В зависимости от используемой модели памяти адрес переменной в памяти может формироваться как 16 : 16 или 16 : 32. Смещение операнда называется его эффективным или исполнительным адресом (Effective Address, EA).

Селектор сегмента можно указать явным или неявным образом. Обычно селектор сегмента загружается в сегментный регистр, а сам регистр выбирается в зависимости от типа выполняемой операции, как показано в табл. 3.2.

**Таблица 3.2.** Критерии выбора сегментного регистра

| Тип операции | Регистр сегмента | Сегмент | Описание |
|---|---|---|---|
| Команды процессора | CS | Программный сегмент | Используется при вызовах команд процессора |
| Обращение к стеку | SS | Сегмент стека | Используется во всех операциях со стеком, а также в операциях с памятью, в которых базовыми являются регистры ESP и EBP |
| Локальные данные | DS | Сегмент данных | Используется во всех операциях с данными, исключая те, в которых применяется стек или строка-приемник при выполнении строковых операций |
| Строки-приемники | ES | Сегмент данных, адресуемый регистром ES | Операнд-приемник в строковых операциях |

Процессор автоматически выбирает сегмент в соответствии с условиями, описанными в табл. 3.2. При сохранении операнда в памяти или загрузке из памяти в качестве сегментного регистра по умолчанию используется DS, но можно и явным образом указать сегментный регистр, применяемый в операции.

Пусть, например, требуется сохранить содержимое регистра EAX в памяти, адресуемой сегментным регистром ES и смещением, находящимся в регистре EBX. В этом случае можно использовать команду

```
mov ES:[EBX]. EAX
```

Обратите внимание на то, что после имени сегмента указывается символ двоеточия.

На уровне процессора замена сегмента задается специальным префиксом замены, который является однобайтовым числом, располагающимся перед кодом команды. В некоторых случаях замена сегмента не допускается:

- для сегмента программного кода — все команды используют исключительно сегментный регистр CS;
- при выполнении строковых операций строка-приемник адресуется только регистром ES;
- операции помещения в стек и извлечения из стека всегда используют регистр SS.

Некоторые инструкции процессора требуют явной инициализации сегментных регистров. В таких случаях селектор сегмента может быть извлечен из 16-разрядного регистра или переменной в памяти, как, например, в следующей команде:

```
mov DS. BX
```

Здесь селектор сегмента, находящийся в регистре BX, помещается в сегментный регистр DS. В некоторых случаях селектор сегмента может определяться через 48-разрядный указатель, находящийся в памяти. При этом младшее двойное слово содержит 32-разрядное смещение, а старшее слово — 16-разрядный селектор сегмента.

В большинстве случаев программисты имеют дело с эффективным адресом операнда, то есть с той частью полного адреса операнда, которая определяет смещение операнда в указанном сегменте. Очень часто термины «эффективный адрес» и «смещение» воспринимаются как синонимы при анализе способов адресации операндов, хотя это не совсем так. Для того чтобы избежать путаницы в дальнейшем, мы будем употреблять более корректный термин «эффективный адрес» вместо термина «смещение», а под смещением понимать числовое значение, которое прибавляется к определенному адресу.

Эффективный адрес операнда, находящегося в памяти, может быть задан несколькими способами. В общем случае мы можем определить эффективный адрес операнда как состоящий из нескольких частей:

- смещения, представляющего собой 8-, 16- и 32-разрядное значение;
- базы, представляющей собой содержимое одного из регистров общего назначения;

- индекса, представляющего собой содержимое одного из регистров общего назначения;
- масштабного множителя, равного 2, 4 или 8.

Эффективный адрес, в общем случае, представляет собой сумму смещения, базы и индекса, причем эта сумма может быть скорректирована с помощью масштабного множителя. Схематически это можно представить так, как показано на рис. 3.11.

| База | | Индекс | | Множитель | | Смещение |
|---|---|---|---|---|---|---|
| EAX<br>EBX<br>ECX<br>EDX<br>ESP<br>EBP<br>ESI<br>EDI | + | EAX<br>EBX<br>ECX<br>EDX<br>EBP<br>ESI<br>EDI | × | 1<br>2<br>4<br>8 | = | 8 бит<br>16 бит<br>32 бита |

EA = База + (Индекс × Множитель) + Смещение

**Рис. 3.11.** Схема вычисления эффективного адреса (EA)

Существуют определенные ограничения, касающиеся применения регистров общего назначения в качестве базовых или индексных при формировании эффективного адреса:

- регистр ESP нельзя использовать в качестве индексного регистра;
- если в качестве базового используется регистр ESP или EBP, то сегментным регистром будет SS. Во всех остальных случаях сегментным регистром по умолчанию является DS.

Следует заметить, что база, индекс и смещение могут применяться в любых комбинациях, причем любой компонент может отсутствовать. Масштабирующий множитель применяется только с индексом. Рассмотрим различные комбинации компонентов для получения эффективного адреса.

**Вариант 1.** Для формирования эффективного адреса используется только смещение. Такую адресацию называют прямой. При этом способе адресации эффективный адрес берется прямо из поля смещения команды и никакие регистры для его вычисления не привлекаются. Этот режим служит для обращения к простым переменным, как показано в примере

```
mov AX, mem1
```

Здесь mem1 — операнд в памяти. Как операнд-источник, так и операнд-приемник должны иметь одинаковый размер, иначе в процессе компиляции будет выдана ошибка. Так, в нашем примере подразумевается, что переменная mem1 определена как слово. Если, например, операнд в памяти является двойным словом, то в команде нужно явным образом указать старшую или младшую часть с помощью оператора PTR:

```
mov AX, word ptr mem1
```

Предположим, что переменная mem1 определена как двойное слово. Тогда показанная ранее команда поместит в регистр AX значение 1D7Fh (рис. 3.12).

| Младший байт | | | Старший байт | |
|---|---|---|---|---|
| 7F | 1D | 34 | EC | mem1 |

**Рис. 3.12.** Размещение переменной mem1 в памяти

Если нужно сохранить в регистре AX значение старшего слова переменной mem1, то следует применить команду

```
mov AX, word ptr mem1+2
```

В этом случае в регистр AX помещаются старшие два байта переменной mem1 (см. рис. 3.12), после чего AX будет содержать значение 0EC34h (обратите внимание на порядок расположения байтов!).

Остановимся более подробно на операторе PTR. В общем случае этот оператор можно представить в виде

*тип* PTR *выражение*

При помощи оператора PTR переменная или метка, задаваемая выражением, может трактоваться как переменная или метка указанного типа. Тип может быть задан одним из имен или значений, показанных в табл. 3.3.

**Таблица 3.3.** Атрибуты оператора PTR

| Тип | Значение |
|---|---|
| BYTE | 1 |
| WORD | 2 |
| DWORD | 4 |
| QWORD | 8 |
| TBYTE | 10 |
| NEAR | 0FFFFh |
| FAR | 0FFFEh |

Выражение может включать в себя любые операнды. Типы BYTE, WORD, DWORD, QWORD и TWORD могут быть использованы только с операндами памяти, а типы NEAR и FAR — только с метками. Если PTR не используется, то ассемблер подразумевает умалчиваемый тип ссылки. Кроме того, оператор PTR служит для организации доступа к объекту, который при другом способе привел бы к ошибке компиляции (например, для доступа к старшему байту переменной размера WORD).

**Вариант 2.** Для формирования эффективного адреса используется только содержимое базового регистра («база»). Такая адресация называется базовой и служит для адресации динамических структур данных, например строк и массивов. Этот способ адресации иногда называют «косвенной адресацией».

Рассмотрим пример программного кода

```
lea BX, mem1
mov AX, [BX]
```

В этом примере mem1 — переменная в памяти размером в слово. Первая команда загружает адрес переменной в регистр BX, а вторая помещает в регистр AX

значение, содержащееся по адресу, который находится в BX. Работу этого фрагмента кода иллюстрирует рис. 3.13.

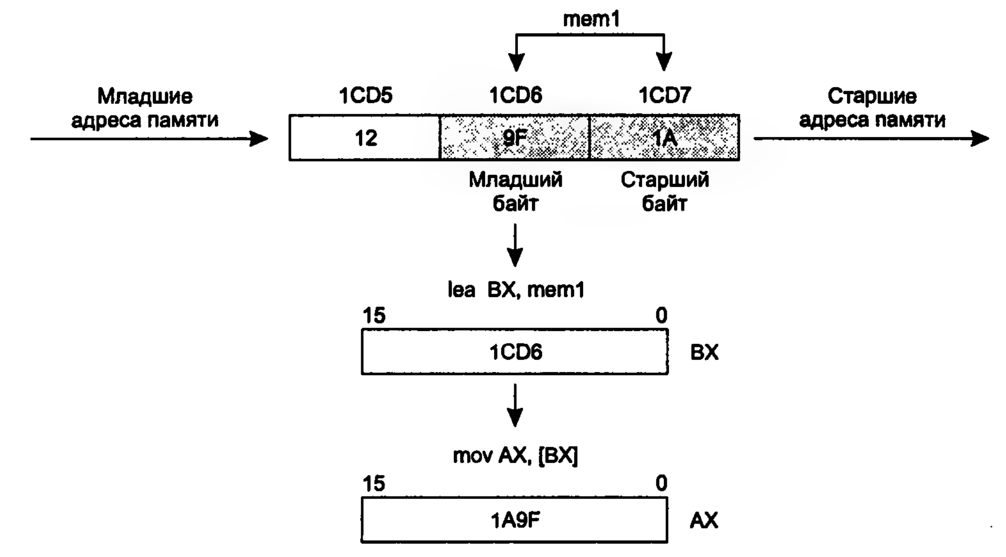



**Рис. 3.13.** Схема косвенной адресации

Предположим, что переменная mem1 размером в слово имеет значение 1A9Fh и занимает два байта в памяти с адресами 1CD6h (младший байт) и 1CD7 (старший байт). После выполнения первой команды в регистре BX будет находиться адрес переменной mem1, являющийся одновременно и адресом первого элемента. Вторая команда помещает в регистры AX значение переменной mem1.

**Вариант 3.** Для формирования эффективного адреса операнда используется содержимое базового регистра плюс смещение («база + смещение»). Такой вариант адресации удобен в двух случаях:

- при доступе к элементам массива, размер которых не кратен 2. В этом случае базовый регистр содержит адрес массива, а смещение позволяет получать доступ к произвольному элементу массива;
- при доступе к полям записей или структур. При этом базовый регистр содержит адрес начала записи или структуры, а смещение определяет элемент, к которому нужно получить доступ. Отдельный, но очень важный случай применения этого варианта — извлечение параметров процедуры из стека посредством регистра EBP, который служит базовым. При этом параметры извлекаются из стека по фиксированным смещениям.

Основное применение базовой адресации — получение доступа к элементам строк и массивов, когда известен начальный адрес данных, а смещение вычисляется в процессе выполнения программы. В макроассемблере MASM можно использовать одну из форм записи:

```
[база + смещение]
[база][смещение]
```

Здесь *база* — регистр, содержащий базовое значение адреса, а *смещение* — значение, которое определяет позицию элемента данных. Вот пример базовой адресации:

```
. . .
s1  DB "String 1"
. . .
lea EBX, s1
mov AL, byte ptr [EBX][5]
. . .
```

Здесь первая команда (lea EBX, s1) помещает в регистр EBX адрес строки, который одновременно является и адресом первого элемента (имеющего индекс 0). Во время выполнения второй команды к содержимому регистра EBX прибавляется значение 5, указывая на 6-й по порядку элемент строки s1 (это символ g), после чего значение этого символа помещается в регистр AL. Таким образом, после выполнения второй команды регистр AL будет содержать символ g. Алгоритм выполнения этого фрагмента программного кода показан на рис. 3.14.

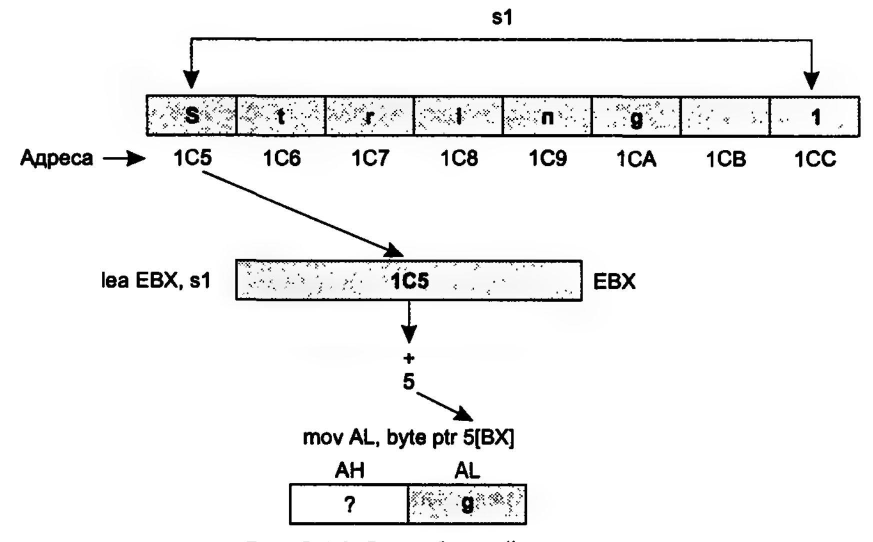



**Рис. 3.14.** Схема базовой адресации

**Вариант 4.** В следующем режиме эффективный адрес формируется по принципу «индекс + смещение». Смещение при таком способе адресации указывает на начало массива чисел или строки, а индексный регистр содержит номер элемента данных. Например, в показанном ниже фрагменте программного кода в регистр AL помещается элемент строки s1 с индексом 10 (11-й элемент строки, символ +):

```
. . .
s1  DB "!@#$%^&*()+][" 
. . .
mov EBX, 10
mov AL, byte ptr s1[EBX]
. . .
```

**Вариант 5.** В следующем режиме эффективный адрес формируется по принципу «(индекс × множитель) + смещение». Множитель обычно используется для доступа к элементам, имеющим размер, кратный 2, например к словам, двойным словам и т. д. Смещение при таком способе адресации указывает на начало массива чисел или строки, а индексный регистр содержит номер элемента данных. Далее показан пример, который демонстрирует этот способ адресации:

```
. . .
s1 DB "0123456789ABCDEF"
. . .
mov   EBX, 7
mov   AL, byte ptr s1[EBX*2]
. . .
```

При указанном значении регистра EBX в регистр AL будет помещен символ E, поскольку он находится по смещению 14 (7 × 2) в строке s1.

**Вариант 6.** В следующем режиме эффективный адрес формируется по принципу «база + индекс + смещение». Такой способ адресации обычно используется для адресации элементов в двухмерных массивах данных или для доступа к отдельным элементам в массивах, содержащих записи. Проанализируем фрагмент программного кода, в котором применяется данный способ адресации:

```
. . .
s1      DB "ABCD EFGH IJKLM"
s2      DB "abcd efgh ijklm"
s3      DB "0123 4567 89"
sarray  label dword
        DD s1
        DD s2
        DD s3
. . .
mov   EBX, sarray+4
mov   ESI, 10
mov   AL, byte ptr [EBX][ESI][2]
. . .
```

Здесь в сегменте данных определен массив строк sarray, содержащий адреса строк s1 – s3. В каждой строке определены группы элементов, разделенные символом пробела. Предположим, нужно получить доступ к символу k, находящемуся в строке s2. Будем использовать регистр EBX как базовый, а регистр ESI как индексный. Поместим в EBX адрес строки s2, где находится искомый элемент (команда mov EBX, sarray+4), а в регистр ESI — смещение группы элементов, в которой находится символ k (величина смещения в данном случае равна 10). Для этого выполним команду

```
mov ESI, 10
```

Символ k находится по смещению 2 относительно группы элементов ijklm, поэтому последняя команда помещает символ в регистр AL:

```
mov AL, byte ptr [EBX][ESI][2]
```

**Вариант 7.** В последнем режиме эффективный адрес формируется по принципу «база + (индекс × множитель) + смещение». Такой способ адресации обычно требу-

ется для адресации элементов в двухмерных массивах данных, когда нужно получить доступ к словам, двойным словам или учетверенным словам. Проанализируем фрагмент программного кода, в котором применяется данный способ адресации:

```
. . .
a1      DD 45, -87, 23, -11, 83, -442, 56, -340
a2      DD 92, -31, 9, -598, 361, 406, -172, 7
a3      DD 234, 8, -177, 921, 380, -12, 0, -51
iarray  label dword
        DD a1
        DD a2
        DD a3
. . .
mov   EBX, iarray+8
mov   ESI, 4
mov   EAX, [EBX][ESI*2][8]
```

Здесь определены три массива целых чисел (a1 – a3), состоящих из двухсловных элементов. Предположим, требуется поместить число 380 (выделенное жирным шрифтом) в регистр EAX. Для этого воспользуемся несколько искусственной схемой, которая поможет понять суть этого метода адресации.

В регистр EBX поместим адрес массива a3 (команда mov EBX, iarray+8), в котором находится искомое число. Таким образом, регистр EBX будет использоваться как базовый. Регистр ESI будет выступать в качестве индексного, куда мы поместим значение 4 (размер двойного слова в байтах) с помощью команды

```
mov ESI, 4
```

Наконец, последняя команда загружает искомый элемент массива a3 (380) в регистр EAX:

```
mov EAX, [EBX][ESI*2][8]
```

В этой команде выражение [ESI*2], равное 8, указывает на элемент массива a3 с индексом 2 (то есть число –177), а выражение [8] определяет смещение на 8. В результате суммирования всех значений (EBX, ESI, 8) получаем эффективный адрес (EA) искомого элемента.

Как видим, базово-индексные способы адресации представляют собой мощный механизм, обеспечивающий удобный доступ к любым структурам данных из программ на языке ассемблера.

Хочу сделать важное замечание: все семь рассмотренных вариантов адресации проверены на компиляторе MASM версии 7.10 из пакета Windows XP DDK.

Рассмотрим еще один способ адресации данных в памяти, который используется в ряде случаев и называется непосредственной адресацией. При этом способе адресации операнд задается непосредственно в инструкции. Например, следующая команда вычитает значение 20 из регистра EAX:

```
sub EAX, 20
```

Все арифметические команды, за исключением команд div и idiv, допускают непосредственную адресацию. Максимальное значение непосредственного операнда варьируется для разных команд, однако в любом случае не может превышать значения, которое может принимать операнд размером в двойное слово без знака ($2^{32}$).

Последний способ адресации, который мы проанализируем, — регистровая адресация. При регистровой адресации операнд находится в регистре общего назначения, а в некоторых случаях — в сегментном регистре. Если команда имеет два операнда, то в большинстве случаев они могут быть регистрами. Вот примеры регистровой адресации:

```
mov   EAX, EDX
add   EAX, ECX
```

Оба операнда должны иметь одинаковую размерность. Следующая команда вызовет ошибку:

```
mov EAX, BL
```

Здесь оба операнда — регистры EAX и BL — имеют разную размерность, поэтому компилятор выдаст ошибку при трансляции этой команды.

Рассмотрим вкратце команды общего назначения процессора Intel Pentium. Более детальный анализ всех групп команд мы будем проводить в следующих главах, когда будут рассматриваться практические аспекты применения языка ассемблера. Команды общего назначения (general-purpose instructions) по функциональному признаку можно разделить на несколько групп:

- команды перемещения (пересылки, передачи) данных;
- команды целочисленной арифметики (сложения, вычитания, умножения и деления);
- команды логических операций;
- команды передачи управления (условных и безусловных переходов, вызовов процедур);
- команды строковых операций (иногда встречается название «строковые, или цепочечные, команды»).

Часть команд сложно отнести к какой-либо группе (например, команды помещения данных в стек или извлечения данных из стека, команды работы с табличными данными и т. д.).

Большинство команд работают с операндами в памяти, адресуемыми одним из способов, рассмотренных ранее, а также с регистрами общего назначения (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP) и с регистрами сегментов (CS, DS, SS, ES, FS, GS).

Макроассемблер MASM версии 6.14 и выше поддерживает все основные команды процессора Intel Pentium, а также специальные группы команд MMX-, SSE- и SSE2-расширений, которые подробно рассматриваются в последующих главах. Перечень всех команд процессора приводится в приложениях А и Б.

# Структура программы на языке ассемблера



Материал этой главы посвящен вопросам организации и компоновки программного кода на языке ассемблера. Затронуты вопросы взаимодействия различных частей ассемблерной программы, организации сегментов программного кода, данных и стека в контексте различных моделей памяти. Напомню, что мы рассматриваем эти аспекты применительно к макроассемблеру MASM фирмы Microsoft, хотя многие положения действительны и для других компиляторов. Начнем с анализа сегментов. Мы уже сталкивались с этими вопросами в главе 3, сейчас же рассмотрим их более детально.

## 4.1. Организация сегментов

Для хорошего понимания, как работает программа на ассемблере, нужно очень четко представлять себе организацию сегментов. Применительно к процессорам Intel Pentium термин «сегмент» имеет два значения:

- Область физической памяти заранее определенного размера. Для 16-разрядных процессоров размер сегмента физической памяти не может превышать 64 Кбайт, в то время как для 32-разрядных может достигать 4 Гбайт.
- Область памяти переменного размера, в которой могут находиться программный код, данные или стек.

Физический сегмент может располагаться только по адресу, кратному 16, или, как иногда говорят, по границе параграфа. Логические сегменты тесно связаны с физическими. Каждый логический сегмент ассемблерной программы определяет именованную область памяти, которая адресуется селектором сегмента, содержащимся в сегментном регистре. Сегментированная архитектура создает определенные трудности в процессе разработки программ. Для небольших программ, меньших 64 Кбайт, программный код и данные могут размещаться в отдельных сегментах, поэтому никаких особых проблем не возникает.

Для больших программ, занимающих несколько сегментов кода или данных, необходимо правильно адресовать данные, находящиеся в разных сегментах данных. Кроме того, если программный код находится в нескольких сегментах, то усложняются реализация переходов и ветвлений в программе, а также вызовы процедур. Во всех этих случаях требуется задавать адреса в виде *сегмент:смещение*.

При использовании 32-разрядного защищенного режима эти проблемы исчезают. Например, в плоской модели памяти (о ней мы поговорим чуть позже) для адресации программного кода и данных достаточно 32-разрядного эффективного адреса внутри непрерывной области памяти.

Логические сегменты могут содержать три основных компонента программы: программный код, данные и стек. Макроассемблер MASM обеспечивает правильное отображение этих компонентов на физические сегменты памяти, при этом сегментные регистры CS, DS и SS содержат адреса физических сегментов памяти.

# 4.2. Директивы управления сегментами и моделями памяти макроассемблера MASM

В макроассемблер MASM включены директивы, упрощающие определение сегментов программы и, кроме того, предполагающие те же соглашения, которые используются в языках высокого уровня Microsoft. Упрощенные директивы определения сегментов генерируют необходимый код, указывая при этом атрибуты сегментов и порядок их расположения в памяти. Везде в этой книге мы будем использовать именно упрощенные директивы определения сегментов, наиболее важные из которых перечислены далее:

- .DATA (.data) — определяет начало инициализированного сегмента данных с именем _DATA и при наличии предыдущего сегмента завершает его. Этой директиве должна предшествовать директива .MODEL. Сегмент, определенный с атрибутом .DATA, должен содержать только инициализированные данные, то есть имеющие начальные значения, например:

```
.data
  val1   DW 11
  string1 DB "Text string"
  byte1  DB ?
```

- .DATA? (.data?) — определяет сегмент данных, в котором располагаются неинициализированные данные. При наличии предыдущего сегмента новый сегмент завершает его. Неинициализированные данные могут объявляться в сегменте .DATA? при помощи оператора ?. Преимуществом директивы .DATA? является то, что при ее использовании уменьшается размер исполняемого файла и, кроме того, обеспечивается лучшая совместимость с другими языками. Этой директиве должна предшествовать директива .MODEL. Вот пример использования директивы .DATA?:

```
.data?
  DB 5 DUP (?)
```

- .CONST (.const) — определяет начало сегмента данных, в котором определены константы. При наличии предыдущего сегмента новый сегмент завершает его. В целях совместимости с другими языками данные должны быть в формате, совместимом с принятыми в языках высокого уровня соглашениями. Сегмент, определенный директивой .CONST, имеет атрибут «только для чтения». Этой директиве должна предшествовать директива .MODEL.
- .STACK (.stack) [*размер*] — определяет начало сегмента стека с указанным *размером* памяти, который должен быть выделен под область стека. Если параметр не указан, размер стека предполагается равным 1 Кбайт. При наличии предыдущего сегмента новый сегмент завершает его. Этой директиве должна предшествовать директива .MODEL.
- .CODE (.code) [*имя*] — определяет сегмент программного кода и заканчивает предыдущий сегмент, если таковой имеется. Необязательный параметр *имя* замещает имя _TEXT, заданное по умолчанию. Если имя не определено, ассемблер создает сегмент с именем _TEXT для моделей памяти tiny, small, compact и flat или сегмент с именем *имя_модуля*_TEXT для моделей памяти medium, large и huge. Этой директиве должна предшествовать директива .MODEL, указывающая модель памяти, используемую программой.
- .MODEL (.model) *модель_памяти* [,*соглашение_о_вызовах*] [,*тип_ОС*] [,*параметр_стека*] — определяет модель памяти, используемую программой. Директива должна находиться перед любой из директив объявления сегментов. Она связывает определенным образом различные сегменты программы, определяемые ее параметрами tiny, small, compact, medium, large, huge или flat. Параметр *модель_памяти* является обязательным.

Если разрабатывается процедура для включения в программу, написанную на языке высокого уровня, то должна быть указана та модель памяти, которая используется компилятором языка высокого уровня. Кроме того, модель памяти должна соответствовать режиму работы (типу) процессора. Это имеет значение для плоской модели памяти, которую можно применять только в режимах .386, .486, .586, .686. Модель памяти определяет, какой тип адресации данных и команд поддерживает программа (near или far). Это имеет смысл для команд перехода, вызовов и возврата из процедур. В табл. 4.1 демонстрируются эти особенности.

**Таблица 4.1.** Параметры моделей памяти

| Модель памяти | Адресация кода | Адресация данных | Операционная система | Чередование кода и данных |
|---|---|---|---|---|
| TINY | NEAR | NEAR | MS-DOS | Допустимо |
| SMALL | NEAR | NEAR | MS-DOS, Windows | Нет |
| MEDIUM | FAR | NEAR | MS-DOS, Windows | Нет |
| COMPACT | NEAR | FAR | MS-DOS, Windows | Нет |
| LARGE | FAR | FAR | MS-DOS, Windows | Нет |
| HUGE | FAR | FAR | MS-DOS, Windows | Нет |
| FLAT | NEAR | NEAR | Windows NT, Windows 2000, Windows XP, Windows 2003 | Допустимо |

Все семь моделей памяти поддерживаются всеми компиляторами MASM, начиная с версии 6.1.

Модель small поддерживает один сегмент кода и один сегмент данных. Данные и код при использовании этой модели адресуются как near (ближние). Модель large поддерживает несколько сегментов кода и несколько сегментов данных. По умолчанию все ссылки на код и данные считаются дальними (far).

Модель medium поддерживает несколько сегментов программного кода и один сегмент данных, при этом все ссылки в сегментах программного кода по умолчанию считаются дальними (far), а ссылки в сегменте данных — ближними (near). Модель compact поддерживает несколько сегментов данных, в которых используется дальняя адресация данных (far), и один сегмент кода с ближней адресацией (near). Модель huge практически эквивалентна модели памяти large.

Должен заметить, что разработчик программ может явно определить тип адресации данных и команд в различных моделях памяти. Например, ссылки на команды внутри одного сегмента кода в модели large можно сделать ближними (near). Проанализируем, в каких случаях лучше всего подходят те или иные модели памяти.

Модель tiny работает только в 16-разрядных приложениях MS-DOS. В этой модели все данные и код располагаются в одном физическом сегменте. Размер программного файла в этом случае не превышает 64 Кбайт. С другой стороны, модель flat предполагает несегментированную конфигурацию программы и используется только в 32-разрядных операционных системах. Эта модель подобна модели tiny в том смысле, что данные и код размещены в одном сегменте, только 32-разрядном. Хочу напомнить, что многие примеры из этой книги разработаны именно для модели flat.

Для разработки программы для модели flat перед директивой .model flat следует разместить одну из директив: .386, .486, .586 или .686. Желательно указывать тот тип процессора, который используется в машине, хотя на машинах с Intel Pentium можно указывать директивы .386 и .486. Операционная система автоматически инициализирует сегментные регистры при загрузке программы, поэтому модифицировать их нужно, только если необходимо смешивать в одной программе 16- и 32-разрядный код. Адресация данных и кода является ближней (near), при этом все адреса и указатели являются 32-разрядными.

Параметр *соглашение_о_вызовах* используется для определения способа передачи параметров при вызове процедуры из других языков, в том числе и языков высокого уровня (C++, Pascal). Параметр может принимать следующие значения: C, BASIC, FORTRAN, PASCAL, SYSCALL, STDCALL. При разработке модулей на ассемблере, которые будут применяться в программах, написанных на языках высокого уровня, обращайте внимание на то, какие соглашения о вызовах поддерживает тот или иной язык. Более подробно соглашения о вызовах мы будем рассматривать при анализе интерфейса программ на ассемблере с программами на языках высокого уровня.

Параметр *тип_ОС* равен OS_DOS, и на данный момент это единственное поддерживаемое значение этого параметра.

Наконец, последний параметр *параметр_стека* устанавливается равным NEARSTACK (регистр SS равен DS, области данных и стека размещаются в одном и том же физическом сегменте) или FARSTACK (регистр SS не равен DS, области данных и стека размещаются в разных физических сегментах). По умолчанию принимается значение NEARSTACK. Рассмотрим примеры использования директивы .MODEL:

```
.model flat, c
```

Здесь параметр flat указывает компилятору на то, что будет использоваться 32-разрядная линейная адресация. Второй параметр с указывает, что при вызове ассемблерной процедуры из другой программы (возможно, написанной на другом языке) будет задействован способ передачи параметров, принятый в языке С. Следующий пример:

```
.model large, c, farstack
```

Здесь используются модель памяти large, соглашение о передаче параметров языка С и отдельный сегмент стека (регистр SS не равен DS).

```
.model medium, pascal
```

В этом примере используются модель medium, соглашение о передаче параметров для Pascal и область стека, размещенная в одном физическом сегменте с данными.

# 4.3. Структура программ на ассемблере MASM

Программа, написанная на ассемблере MASM, может состоять из нескольких частей, называемых модулями, в каждом из которых могут быть определены один или несколько сегментов данных, стека и кода. Любая законченная программа на ассемблере должна включать один главный, или основной (main), модуль, с которого начинается ее выполнение. Основной модуль может содержать программные сегменты, сегменты данных и стека, объявленные при помощи упрощенных директив. Кроме того, перед объявлением сегментов нужно указать модель памяти при помощи директивы .MODEL. Поскольку подавляющее большинство современных приложений являются 32-разрядными, то основное внимание в этом разделе мы уделим именно таким программам, хотя не обойдем вниманием и 16-разрядные программы, которые все еще используются. Начнем с 16-разрядных программ.

В следующем примере показана 16-разрядная программа на ассемблере, в которой используются упрощенные директивы ассемблера MASM:

```
.model small, c  ; эта директива указывается до объявления
                 ; сегментов
.stack 100h      ; размер стека 256 байт
.data            ; начало сегмента данных
. . .
; данные
. . .
.code            ; здесь начинается сегмент программ
main:
. . .
```

```
; команды ассемблера
. . .
end main
end
```

Здесь оператор end main указывает на точку входа main в главную процедуру. Оператор end закрывает последний сегмент и обозначает конец исходного текста программы. В 16-разрядных приложениях MS-DOS можно инициализировать сегментные регистры так, чтобы они указывали на требуемый логический сегмент данных. Листинг 4.1 демонстрирует это.

**Листинг 4.1.** Пример адресации сегментов в программе MS-DOS

```
.model large
.data
 s1 DB "TEST STRING$"
.code
 mov AX, @data
 mov DS, AX
 lea DX, s1
 mov AH, 9h
 int 21h
 mov ax, 4c00h
 int 21h
end
```

Здесь на экран дисплея выводится строка s1. При помощи следующих команд в сегментный регистр DS помещается адрес сегмента данных, указанного директивой .data:

```
mov AX, @data
mov DS, AX
```

Затем строка s1, адресуемая через регистры DS:DX, выводится на экран с использованием прерывания 9h функции 21h MS-DOS. Попробуйте закомментировать проанализированные две строки кода и посмотреть на результат работы программы.

Для 32-разрядных приложений шаблон исходного текста выглядит иначе:

```
.model flat
.stack
.data
; данные
.code
main:
. . .
; команды ассемблера
. . .
end main
end
```

Основное отличие от предыдущего примера — другая модель памяти (flat), предполагающая 32-разрядную линейную адресацию с атрибутом near.

Как видно из примера, «классический» шаблон 32-разрядного приложения содержит область данных (определяемую директивой .data), область стека (директива .stack) и область программного кода (директива .code). Может случиться

так, что 32-разрядному приложению на ассемблере потребуется несколько отдельных сегментов данных и/или кода. В этом случае разработчик может создать их с помощью директивы SEGMENT. Директива SEGMENT определяет логический сегмент и может быть описана следующим образом:

```
имя SEGMENT список атрибутов
. . .
имя ENDS
```

Замечу, что директива SEGMENT может применяться с любой моделью памяти, не только flat. При использовании директивы SEGMENT потребуется указать компилятору на то, что все сегментные регистры устанавливаются в соответствии с моделью памяти flat. Это можно сделать при помощи директивы ASSUME:

```
ASSUME CS:FLAT, DS:FLAT, SS:FLAT, ES:FLAT, FS:ERROR, GS:ERROR
```

Регистры FS и GS программами не используются, поэтому для них указывается атрибут ERROR.

Сейчас мы рассмотрим программный код 32-разрядной процедуры на ассемблере (она называется _seg_ex), в которой используются два логических сегмента данных. Процедура выполняет копирование строки src, находящейся в сегменте данных data1, в область памяти dst в сегменте данных data2 и содержит один логический сегмент программного кода (code segment).

Успокою читателей, незнакомых с принципами работы процедур (они рассмотрены далее в книге): в данном случае нас будет интересовать код внутри процедуры _seg_ex (команды, находящиеся между директивами _seg_ex proc и _seg_ex endp). Исходный текст программного кода процедуры _seg_ex представлен в листинге 4.2.

**Листинг 4.2.** Использование двух логических сегментов данных в 32-разрядной процедуре

```
.586
.model flat
 option casemap:none
data1 segment
  src  DB   "Test STRING To Copy"
  len  EQU  $-src
data1  ends
data2  segment public
  dst  DB  len+1 DUP('+')
data2  ends
code segment
_seg_ex proc
 assume CS:FLAT,DS:FLAT, SS:FLAT, ES:FLAT, FS:ERROR, GS:ERROR
 mov    ESI, offset data1
 mov    EDI, offset data2
 cld
 mov    CX, len
 rep    movsb
 mov    EAX, offset data2
 ret
_seg_ex endp
code ends
end
```

При использовании модели flat доступ к данным осуществляется по 32-разрядному смещению, поэтому смысл показанных ниже команд, загружающих адреса логических сегментов (а заодно и адреса строк src и dst) в регистры ESI и EDI, думаю, понятен:

```
mov ESI, offset data1
mov EDI, offset data2
```

Группа следующих команд выполняет копирование строки src в dst, при этом регистр CX содержит количество копируемых байтов:

```
cld
mov CX, len
rep movsb
```

В регистре EAX возвращается адрес строки-приемника dst. Обращаю внимание читателей на то, что никакой инициализации сегментных регистров не требуется, поскольку это делается при помощи директивы .model flat. Еще один важный момент: программа, использующая модель flat, выполняется в одном физическом сегменте, хотя логических сегментов может быть несколько, как в нашем случае.

Работоспособность процедуры легко проверить, вызвав ее из программы на Visual C++ .NET (нужно только включить объектный файл процедуры в проект приложения). Исходный текст приложения приведен в листинге 4.3.

**Листинг 4.3.** Программа, вызывающая процедуру seg_ex

```
#include <stdio.h>
extern "C" char* seg_ex(void);
int main(void)
{
 printf("EXTERNAL MODULE EXAMPLE: %s\n", seg_ex());
 return 0;
}
```

Здесь процедура seg_ex является внешней, поэтому объявлена как extern.

Результатом выполнения программы будет строка на экране дисплея

```
EXTERNAL MODULE EXAMPLE: Test STRING To Copy+
```

# Организация вычислительных циклов



Большинство программ, независимо от того, на каком языке они написаны, в процессе работы требуют изменения линейной последовательности выполнения операторов и перехода на другие ветви программного кода. В языках высокого уровня, таких, например, как C++ или Pascal, существуют специальные операторы, позволяющие выполнять ветвление программ, в то время как на языке ассемблера для организации переходов требуется писать специальный код.

Подобные операторы или группы операторов называют логическими структурами. Принципиально, существует два типа логических структур, на основе которых можно создать все остальные логические выражения и вычислительные алгоритмы:

- Структура, основанная на сравнении каких-либо величин и выборе варианта продолжения программы в зависимости от результата сравнения. Такую логическую структуру можно описать выражением «если А, то В, иначе С». В языках высокого уровня ее аналогом является оператор `if ... else`. Кроме того, существуют и расширенные варианты оператора `if ... else`, например `switch ... case`.
- Структура, в основе которой лежит алгоритм повторяющихся вычислений. Такой алгоритм выполняется при соблюдении определенных условий, которые проверяются в начале или в конце каждой итерации. Такие логические структуры в языках высокого уровня реализованы как операторы `while`, `do ... while`, `for`, `repeat ... until` и т. д.

Логические структуры языков высокого уровня придают особую гибкость и мощь этим языкам и позволяют относительно легко строить самые сложные алгоритмы. А как обстоят дела с языком ассемблера, ведь здесь нет подобных операторов? Несмотря на отсутствие подобных операторов, ассемблер обладает довольно мощными командами, позволяющими строить любые сколь угодно сложные логические структуры, в том числе и такие, которые довольно трудно, а иногда и невозможно реализовать в языках высокого уровня.

Хочу сразу же оговориться: мы не будем брать во внимание псевдомакросы логических структур, такие, как .IF и .WHILE, включенные в популярные языки ассемблера (в данном случае MASM). Псевдомакросы имитируют логические структуры высокого уровня, например if … else, но значительно уступают им по возможностям, хотя и улучшают читабельность кода. Если внимательно посмотреть на программный код макросов, то вы увидите, что они содержат команды ассемблера, которые вполне можно написать самостоятельно. Кроме того, псевдомакросы скрывают механизм операции, что может быть полезно для программирования, но не для изучения алгоритмов функционирования. По этим причинам я сознательно отказался от использования псевдомакросов и макрорасширений ассемблера при рассмотрении программного кода.

Посмотрим, как можно создать логические структуры на языке ассемблера и использовать их в программах. В подавляющем большинстве случаев для создания логических структур и выражений на ассемблере, таких, например, как if … else, while и так далее, требуется два условия:

- установить один или несколько битов в регистре флагов процессора;
- проверить состояние бита (группы битов) с помощью команд условного перехода и передать управление на соответствующую ветвь программы.

Большинство команд ассемблера после выполнения устанавливают биты регистра флагов определенным образом, поэтому у программиста есть довольно широкий выбор возможностей для организации вычислительных алгоритмов и логических структур. Естественно, необходимо учитывать особенности выполнения той или иной команды, поскольку можно получить совершенно неожиданные побочные эффекты.

Для передачи управления можно использовать и команды безусловного перехода; мы рассмотрим такую возможность позже.

Таким образом, для построения вычислительных алгоритмов необходимо анализировать состояние битов регистра флагов EFLAGS центрального процессора. Девять из 16 бит регистра являются активными и определяют текущее состояние машины и результатов выполнения команд. Многие арифметические команды и команды сравнения изменяют состояние флагов. Напомню назначение отдельных битов регистра флагов:

- CF (Carry Flag — флаг переноса) — содержит значение переносов (0 или 1) из старшего разряда при арифметических операциях и некоторых операциях сдвига и циклического сдвига;
- PF (Parity Flag — флаг четности) — проверяет младшие 8 бит результатов выполнения операций над данными. Нечетное число битов приводит к установке этого флага в 0, а четное — в 1. Не следует путать флаг четности с битом контроля на четность;
- AF (Auxiliary Carry Flag — дополнительный флаг переноса) — устанавливается в 1, если арифметическая операция приводит к переносу четвертого справа бита (бит 3) в регистровой однобайтовой команде. Данный флаг имеет отношение к арифметическим операциям над ASCII-символами и к полям, содержащим десятичные упакованные числа;

- ZF (Zero Flag — флаг нуля) — устанавливается после выполнения арифметических команд и команд сравнения. Ненулевой результат операции приводит к установке этого флага в 0, а нулевой — к установке флага в 1. Этот флаг проверяется при помощи команд условного перехода je и jz;
- SF (Sign Flag — знаковый флаг) — устанавливается в соответствии со знаком результата (старшего бита) после выполнения арифметических операций: при положительном результате флаг устанавливается в 0, а при отрицательном — в 1. Этот флаг проверяется при помощи команд условного перехода jg и jl;
- TF (Trap Flag — флаг пошагового выполнения) — если этот флаг установлен в 1, то процессор переходит в режим пошагового выполнения команд, то есть в каждый момент выполняется одна команда под управлением пользователя;
- IF (Interrupt Flag — флаг прерывания) — при нулевом состоянии этого флага прерывания запрещены, а при единичном — разрешены;
- DF (Direction Flag — флаг направления) — используется в строковых операциях для определения направления передачи данных. При нулевом состоянии команда увеличивает содержимое регистров SI (ESI) и DI (EDI), а при ненулевом уменьшает содержимое этих регистров;
- OF (Overflow Flag — флаг переполнения) — фиксирует арифметическое переполнение, то есть перенос в старший бит или из старшего (знакового) бита при знаковых арифметических операциях.

Чаще всего в программах используются флаг переноса CF, флаг знака SF, флаг нуля ZF, немного реже — флаг четности PF, флаг направления DF, флаг переполнения OF и флаг дополнительного переноса AF. Еще реже, преимущественно в специальных случаях, используются флаги TF и IF.

Рассмотрим некоторые примеры, демонстрирующие методику выполнения условных переходов в программах.

# 5.1. Условные переходы и ветвления

Организацию ветвлений в программах на ассемблере лучше всего объяснить на примере. В следующем фрагменте программного кода выполняется переход на метку next при равенстве нулю содержимого регистра ECX. Равенство нулю содержимого ECX определяется при помощи команды cmp, которая воздействует на флаги AF, CF, OF, PF, SF и ZF:

```
   . . .
   cmp  ECX, 0
   jz   next
     обработка ситуации, когда ECX не равен 0
next:
     обработка ситуации, когда ECX равен 0
   . . .
```

Если ECX содержит нулевое значение, то команда cmp устанавливает флаг нуля ZF в единицу. Команда jz проверяет флаг ZF и, если он равен 1, передает управление на адрес, указанный в ее операнде, то есть на метку next. Фактически данный фрагмент программного кода реализует логическую структуру if, анализирующую условие ECX = 0.

Этот пример демонстрирует один из типичных вариантов организации ветвлений с использованием команды cmp. В данном случае эта команда устанавливает или сбрасывает флаг ZF, в зависимости от равенства или неравенства нулю содержимого регистра ECX. Состояние флага анализируется командой jz next, после чего осуществляется переход на одну из двух возможных ветвей программного кода. Большинство команд процессоров Intel воздействуют на флаги, что позволяет задействовать их для организации довольно сложных вычислительных алгоритмов.

Наиболее часто для организации ветвлений используются команды сравнения (cmp, test), а также арифметические (add, sub и др.) и логические команды (and, or, xor). Например, команда test выполняет операцию логического «И» над двумя операндами и в зависимости от результата устанавливает флаги SF, ZF и PF. При этом флаги OF и CF сбрасываются, а флаг AF имеет неопределенное значение. Очень важно то, что команда test не изменяет ни одного из операндов. Ее очень удобно использовать для анализа отдельных битов сравниваемых величин, как в этом примере:

```
. . .
test  AX. 1
jne   bit1_set
. . .
```

Здесь анализируется нулевой бит регистра AX. Если он установлен в 1, то флаг ZF устанавливается в 0 и выполняется переход на метку bit1_test.

Ассемблер поддерживает большое количество команд условного перехода, которые осуществляют передачу управления в зависимости от состояний регистра флагов. При этом следует учитывать некоторые особенности использования таких команд. В подавляющем большинстве случаев команды условных переходов оперируют флагами, установленными в результате сравнения числовых величин. Типы данных, над которыми выполняются арифметические операции и операции сравнения, определяют выбор команд: беззнаковые или знаковые. Типичными примерами беззнаковых данных являются символьные строки, имена, адреса и натуральные числа. Многие числовые значения могут быть как положительными, так и отрицательными.

Например, если регистр AX содержит 11000110B, а BX — 00010110B, то для беззнаковых данных значение в AX будет больше BX, а для знаковых — меньше. Перечень команд условных переходов для беззнаковых данных приведен в табл. 5.1.

**Таблица 5.1.** Команды условных переходов для чисел без знака

| Мнемоника | Описание | Проверяемые флаги |
|---|---|---|
| JE/JZ | Переход, если равно/нуль | ZF |
| JNE/JNZ | Переход, если не равно/не нуль | ZF |

| Мнемоника | Описание | Проверяемые флаги |
|---|---|---|
| JA/JNBE | Переход, если выше/не ниже или равно | ZF, CF |
| JAE/JNB | Переход, если выше или равно/не ниже | CF |
| JB/JNAE | Переход, если ниже/не выше или равно | CF |
| JBE/JNA | Переход, если ниже или равно/не выше | CF, AF |

Любую проверку можно выполнить с помощью одного из двух мнемонических кодов. Например, команды jb и jnae генерирует один и тот же объектный код, хотя мнемоническое обозначение команды jb понять легче, чем jnae.

Перечень команд условных переходов для знаковых данных приведен в табл. 5.2.

**Таблица 5.2.** Команды условных переходов для чисел со знаком

| Мнемоника | Описание | Проверяемые флаги |
|---|---|---|
| JE/JZ | Переход, если равно/нуль | ZF |
| JNE/JNZ | Переход, если не равно/не нуль | ZF |
| JG/JNLE | Переход, если больше/не меньше или равно | ZF, SF, OF |
| JGE/JNL | Переход, если больше или равно/не меньше | SF, OF |
| JL/JNGE | Переход, если меньше/не больше или равно | SF, OF |
| JLE/JNG | Переход, если меньше или равно/не больше | ZF, SF, OF |

Обратите внимание на то, что команды перехода для условий равно или нуль (je/jz) и не равно или не нуль (jne/jnz) присутствуют в обеих таблицах для беззнаковых и знаковых данных. Состояние равно/нуль не зависит от знака числа.

Помимо проверок на равенство-неравенство операндов, очень часто требуется анализировать и другие флаги. Все такие проверки представлены в табл. 5.3.

**Таблица 5.3.** Команды условных переходов для специальных проверок

| Мнемоника | Описание | Проверяемые флаги |
|---|---|---|
| JS | Переход, если число отрицательно | SF |
| JNS | Переход, если число положительно | SF |
| JC | Переход, если есть перенос | CF |
| JNC | Переход, если нет переноса | CF |
| JO | Переход, если есть переполнение | OF |
| JNO | Переход, если нет переполнения | OF |
| JP/JPE | Переход, если есть паритет | PF |
| JNP/JP | Переход, если нет паритета | PF |

Еще одна команда условного перехода проверяет равенство содержимого регистра CX нулю. Эта команда необязательно должна располагаться непосредственно за арифметической командой или командой сравнения. Команда jcxz может быть помещена в начало цикла, где она проверяет содержимое регистра CX.

# 5.2. Команда безусловного перехода jmp

При выполнении команды безусловного перехода jmp, независимо от состояния флагов процессора, программа продолжает выполняться с новой ветви. При этом новый адрес команды загружается в регистр-счетчик команд EIP и выполнение программного кода продолжается с этого адреса.

В языке ассемблера новое место, откуда продолжается выполнение программы, в большинстве случаев обозначается меткой, которую процессор преобразует в исполнительный адрес. Если переход происходит в текущий сегмент, то смещение метки загружается непосредственно в регистр-счетчик команд EIP. Если же метка находится в другом сегменте кода, то адрес сегмента дополнительно загружается в регистр CS. Для преобразования метки в вид *сегмент:смещение* используются три формата команды jmp:

```
jmp short целевой_адрес
jmp near  ptr целевой_адрес
jmp far   ptr целевой_адрес
```

Здесь *целевой_адрес* — адрес команды, которая будет выполняться после перехода. Вот несколько примеров команды jmp:

```
jmp label1   ; адрес команды, которая будет выполняться при
             ; переходе, находится в текущем сегменте команд
jmp near ptr label1 ; адрес следующей команды находится
                    ; в текущем сегменте команд
jmp short label1   ; адрес команды, которая будет выполняться
                   ; при переходе, находится в диапазоне
                   ; -128 - +127
jmp far ptr label1  ; адрес команды, которая будет
                    ; выполняться при переходе, находится
                    ; в другом сегменте
```

Рассмотрим операторы, указанные перед целевым адресом. Оператор short указывает на то, что нужно сделать переход на метку в диапазоне от –128 до +127, начиная от адреса следующей команды. В этом случае к содержимому регистра указателя команд EIP прибавляется 8-разрядное целое число.

Оператор near ptr указывает на метку в текущем сегменте, при этом к регистру указателя команд EIP прибавляется 16-разрядное смещение. Наконец, оператор far ptr указывает, что необходимо сделать переход на метку в другом сегменте. В этом случае сегментная часть адреса метки загружается в регистр CS, а смещение — в EIP.

Рассмотренные нами модификации команды jmp являются классическими для 16-разрядных приложений и берут свое начало от «времен MS-DOS», когда отдельный сегмент кода не мог использовать пространство памяти больше чем 64 Кбайта, а для создания больших программ требовалось определенным образом компоновать несколько сегментов кода и данных.

Любое современное приложение является 32-разрядным и оперирует с линейным пространством адресов размером до 4 Гбайт. При разработке ассемблерных программ, как упоминалось в главе 3, используется модель памяти flat, а это

означает, что программа занимает непрерывную область адресов, в которой размещаются данные и код. По этой причине любые команды адресуются 32-разрядным смещением в пространстве адресов программы.

При запуске 32-разрядного приложения все сегментные регистры устанавливаются в одно и то же значение. Для программистов, работающих с программами в DOS, 32-разрядное Windows-приложение может напоминать COM-файл, поскольку в таком файле можно работать только со смещениями. В 32-разрядных приложениях все метки и переходы считаются ближними (near ptr) в диапазоне адресов 4 Гбайт.

Команду jmp можно использовать не только для безусловного перехода в сегменте программного кода, но и для организации ветвлений. Для этого можно применить один из ее форматов, показанных далее:

```
jmp reg16
jmp reg32
jmp word ptr [reg16]
jmp dword ptr [reg32]
```

Здесь *reg16* (*reg32*) — один из 16- или 32-разрядных регистров. Для первых двух форматов команд из списка адрес, по которому передается управление, должен находиться в одном из этих регистров.

Если используется 32-разрядный регистр (*reg32*), то адрес команды, на которую передается управление, также является 32-разрядным. Этот формат команды jmp характерен для 32-разрядных Windows-приложений.

Последние два формата команды jmp используют механизм косвенной адресации, при этом регистр содержит адрес ячейки памяти, в которой находится адрес команды, получающей управление. Проиллюстрируем вышеизложенное примерами:

```
 . . .
 .code
   . . .
L1:
 xor EDX, EDX
 . . .
 lea ESI, L1
 jmp ESI
  . . .
```

В этом примере в регистр ESI помещается смещение метки L1, после чего с помощью команды jmp ESI управление передается на эту метку.

```
 . . .
 .data
   label_offset DD L1
 .code
   . . .
L1:
 xor EDX, EDX
 . . .
 lea ESI, label_offset
 jmp dword ptr [ESI]
  . . .
```

В этом примере в регистр ESI помещается адрес переменной label_offset, в то время как сама переменная label_offset содержит адрес метки L1. Команда jmp dword ptr [ESI] в этом случае передает управление на метку L1.

Как видно из примеров, использование в качестве операндов регистров или ячеек памяти придает команде безусловного перехода большую гибкость, чем применение прямого смещения, что позволяет создавать ветвления и переходы в программе. Далее мы рассмотрим несколько примеров таких ветвлений.

Следующее 16-разрядное приложение, исходный текст которого показан в листинге 5.1, выводит на экран строки s1, s2 и s3.

**Листинг 5.1.** Вывод трех символьных строк на экран

```
.model small
.stack 100h
.data
  s1  DB 0dh, 0ah, "String 1$"
  s2  DB 0dh, 0ah, "String 2$"
  s3  DB 0dh, 0ah, "String 3$"

  sarray   label word          ; массив, в котором хранятся адреса строк
      DW s1                    ; s1 и s2
      DW s2
      DW s3
  num DW 0                     ; индекс в адресе перехода команды jmp
  label_array label word       ; массив адресов меток
      DW L1                    ; адрес метки L1
      DW L2                    ; адрес метки L2
      DW L3                    ; адрес метки L3
.code
 start:
  mov   AX, @data
  mov   DS, AX
  mov   ES, AX
  ;
  mov   CX, 3                  ; счетчик цикла -> CX
  lea   DI, label_array        ; адрес массива меток
next:
  mov   SI, DI
  mov   BX, num                ; индекс перехода -> BX
  shl   BX, 1                  ; умножить на 2 для правильной адресации
                               ; меток в массиве label_array
  add   SI, BX                 ; сформировать адрес перехода
                               ; для команды jmp
  jmp   word ptr [SI]          ; перейти по адресу, находящемуся
                               ; в регистре SI (L1 или L2)
wedge:
  inc   num                    ; инкремент индекса переходов
  loop  next                   ; повторить цикл
  ;
 L1:                           ; фрагмент кода при переходе на метку L1
  lea  DX, s1
  mov  AH, 9h
  int  21h
```

```
 jmp  wedge                   ; вернуться в цикл
L2:                           ; фрагмент кода при переходе на метку L2
 lea  DX, s2
 mov  AH, 9h
 int  21h
 jmp  wedge
L3:                           ; фрагмент кода при переходе на метку L3
 lea  DX, s3
 mov  AH, 9h
 int  21h
 ;
 mov  AH, 1h                  ; ожидать ввода любого символа
 int  21h
 ;
 mov  AX, 4c00h               ; завершение программы
 int  21h
 end  start
 end
```

В этой программе продемонстрирована техника использования команды безусловного перехода jmp для организации трех ветвлений по адресам, определяемым метками L1, L2 и L3. Адрес перехода команды jmp формируется в регистре SI следующим образом: вначале в SI загружается базовый адрес массива меток label_array, после чего к нему прибавляется смещение, кратное двум (метки L1 – L3 имеют двухбайтовый адрес). Затем из сформированного таким образом адреса извлекается смещение одной из меток и выполняется переход на соответствующую ветвь программы. Например, для получения смещения метки L2 необходимо к адресу label_array прибавить значение 2 (индекс num = 1). После выполнения программы на экране должны отобразиться строки:

```
String 1
String 2
String 3
```

Как видно из примера, команду безусловного перехода jmp можно применить для организации ветвлений в программе в зависимости от значения каких-либо параметров. Рассмотрим еще один, довольно сложный пример, в котором команда jmp используется для организации ветвлений и фактически моделируется логическая структура высокого уровня switch ... case языка C++ (или оператор case языка Pascal), обладающая очень большими вычислительными возможностями. В языке ассемблера довольно сложно реализовать такую структуру, и один из вариантов реализации, который мы рассмотрим, базируется на использовании команды jmp.

Пример представляет собой 32-разрядную процедуру (она называется _case_1). В качестве входного параметра процедура принимает целое число из диапазона 0–2, а в регистре EAX возвращает адрес строки, соответствующий значению параметра. Принципы организации процедур мы рассмотрим в следующих главах, сейчас же акцентируем наше внимание на работе программного кода процедуры _case_1, не вникая в детали ее взаимодействия с другими частями программы.

Для извлечения единственного параметра используется регистр EBP, а сам параметр для выполнения дальнейших манипуляций помещается в регистр EBX. Исходный текст процедуры представлен в листинге 5.2.

**Листинг 5.2.** Ассемблерный аналог конструкции case

```
.686
.model flat
option casemap: none
.data
  s1  DB "String 1", 0
  s2  DB "String 2", 0
  s3  DB "String 3", 0
  err DB "Incorrect parameter!", 0
  label_array label dword            ; массив меток, в котором будут
                                     ; находиться смещения
                                     ; меток L1, L2 и L3
     DD 3 DUP (?)
.code
 _case_1 proc
   push   EBP
   mov    EBP, ESP
   mov    EBX, dword ptr [EBP+8]     ; извлекаем параметр (номер строки)
                                     ; и сохраняем его в регистре EBX
   lea    ESI, label_array           ; адрес массива меток -> ESI
   mov    [ESI], offset L1           ; заполняем массив меток смещениями
   mov    [ESI+4], offset L2         ; меток L1, L2 и L3
   mov    [ESI+8], offset L3
   lea    EAX, err_exit              ; сохраняем в регистре EAX смещение
                                     ; метки для выхода из процедуры
                                     ; в случае ошибки
   shl    EBX, 2                     ; поскольку для адресации
                                     ; используются двойные слова,
                                     ; умножаем номер строки на 4
   cmp    EBX, 8                     ; значение учетверенного параметра
                                     ; не должно превышать 8 (номер строки
                                     ; лежит в диапазоне 0-2)
   jle    next1                      ; верхнее значение меньше 8? Если
                                     ; да, следующая проверка
   jmp    EAX                        ; нет, параметр превышает значение 2,
                                     ; выйти из процедуры с ошибкой
 next1:
   cmp    EBX, 0                     ; параметр не является отрицательным
                                     ; числом? Если
   jge    get_string                 ; нет, продолжить выполнение
                                     ; процедуры
   jmp    EAX                        ; да, параметр вне диапазона, выйти
                                     ; с ошибкой
 get_string:                         ; параметр находится в нужном
                                     ; диапазоне, получить адрес
                                     ; соответствующей строки и выйти из
                                     ; процедуры
   cmovge EAX, [ESI][EBX]
   jmp    EAX
 L1:                                 ; сюда передается управление при
                                     ; значении входного параметра,
                                     ; равном 0
```

```
  lea    EAX. s1                      ; адрес строки s1 -> EAX
  jmp    exit                         ; выход из процедуры
L2:                                   ; сюда передается управление при
                                      ; значении входного параметра.
                                      ; равном 1
  lea    EAX. s2                      ; адрес строки s2 -> EAX
  jmp    exit                         ; выход из процедуры
L3:                                   ; сюда передается управление при
                                      ; значении входного параметра.
                                      ; равном 2
  lea    EAX. s3                      ; адрес строки s3 -> EAX
  jmp    exit                         ; выход из процедуры
err_exit:                             ; сюда передается управление
                                      ; при возникновении ошибки
  lea    EAX. err                     ; адрес сообщения об ошибке -> EAX
exit:
  pop    EBP
  ret
_case_1   endp
end
```

Анализ работы процедуры начнем со строк

```
lea    ESI. label_array
mov    [ESI]. offset L1
mov    [ESI+4]. offset L2
mov    [ESI+B]. offset L3
```

Как и в предыдущем примере, вначале заполняем массив меток смещениями используемых ветвей программы. Поскольку 32-разрядные приложения работают со смещениями, равными двойному слову, то наш массив label_array состоит из трех двойных слов, в которых и сохраняются смещения меток L1, L2 и L3. Все эти действия и выполняют четыре команды, показанные выше.

Следующая команда помещает в регистр EAX смещение метки, куда должно передаваться управление в случае ошибки:

```
lea    EAX. err_exit
```

Для передачи управления в нашей процедуре используется команда

```
jmp EAX
```

Она принимает в качестве операнда регистр (в данном случае — EAX), содержащий смещение команды, куда передается управление.

С помощью следующей команды устанавливается смещение одной из меток (L1, L2 или L3), в которую должно передаваться управление при корректном значении параметра процедуры:

```
shl    EBX. 2
```

Фрагмент программного кода, в котором выполняется проверка параметра на принадлежность диапазону 0–2, думаю, понятен и в объяснениях не нуждается. Если полученный параметр корректен, то выполняется команда

```
cmovge EAX. [ESI][EBX]
```

Остановимся на работе этой инструкции ассемблера более подробно. Описание группы команд, к которой принадлежит cmovge, приводится далее в этой главе, но в нашем случае эта инструкция выполняет две функции:

- анализирует результат предыдущей операции (флаг SF);
- если SF = 1, то в регистр EAX помещается смещение одной из меток (L1, L2 или L3). Само смещение находится по адресу, равному сумме адресов массива label_array (регистр ESI) и индекса строки (регистр EBX).

Подобную процедуру при желании можно усовершенствовать и использовать для организации ветвлений в программах на ассемблере.

## 5.3. Организация циклов

Очень часто условные переходы используются при программировании циклических операций, или циклов, когда обрабатывается группа элементов. Количество итераций (прохождений) в цикле чаще всего определяется количеством обрабатываемых элементов, хотя это и не обязательно. Цикл может закончиться в одном из двух случаев:

- выполнены все итерации;
- обнаружено условие, согласно которому должен произойти выход из цикла.

Рассмотрим следующий пример: пусть необходимо подсчитать количество символов в строке. Условимся, что такая строка оканчивается нулем, и будем использовать этот факт как признак конца цикла. Вот фрагмент программного кода, реализующий этот алгоритм:

```
 . . .
.data
  s1   DB "ABCDFEG". 0
.code
  . . .
  mov  AL. 0
  lea  SI. s1
next:
  cmp  byte ptr [SI]. 0
  je   exit
  inc  SI
  inc  AL
  jmp   next
exit:
  . . .
```

Проанализируем этот фрагмент кода. В качестве счетчика элементов используется регистр AL, в который перед началом вычислений помещается 0. Для анализа элемента на равенство нулю нам понадобится адрес строки или, что одно и то же, адрес первого элемента строки. Значение адреса помещается в регистр SI. Таким образом, к элементу строки можно получить доступ по его адресу, определяемому парой регистров DS : SI. В каждой итерации анализируется признак конца строки с помощью команды

```
cmp byte ptr [SI]. 0
```

Если признак конца строки обнаружен, то происходит выход из цикла. Если элемент строки не равен 0, то к счетчику элементов в регистре AL прибавляется 1, а в регистр SI загружается адрес следующего элемента строки при помощи команды

```
inc SI
```

Далее цикл повторяется. Как видно из примера, цикл заканчивается по условию (достигнут конец строки).

Если известно заранее количество итераций в цикле, то признаком окончания цикла является выполнение всех итераций. В следующем примере подсчитывается количество вхождений символа A в строку s1. Размер строки определяется константой len, поэтому можно использовать это значение для инициализации счетчика цикла:

```
. . .
.data
 s1   DB  "ABCAEFGAGEBA"
 len  EQU $-s1
.code
 . . .
 mov  DX, len
 mov  AL, 'A'
 xor  BL, BL
 lea  SI, s1
next:
 cmp  byte ptr [SI], AL
 je   inc_counter
continue:
 dec  DX
 jz   exit
 inc  SI
 jmp  next
inc_counter:
 inc  BL
 jmp  continue
  . . .
exit:
 . . .
```

Посмотрим, как работает этот код. Поскольку количество итераций заранее известно и равно len, можно загрузить это значение в регистр DX и по окончании каждой итерации уменьшать содержимое DX на 1. Выход из цикла произойдет при значении DX, равном 0. Количество обнаруженных в строке символов A запоминается в счетчике символов, в качестве которого используется регистр BL (начальное значение равно 0).

В самом цикле выполняется сравнение значения текущего символа с содержимым регистра AL. Если обнаружено совпадение, то есть проверяемый элемент равен A, то регистр BL инкрементируется:

```
  cmp  byte ptr [SI], AL
  je   inc_counter
   . . .
inc_counter:
  inc  BL
```

В нашем последнем примере использовался счетчик цикла на регистре BL. В языке ассемблера для организации циклов с заранее определенным количеством итераций очень удобно применять команду loop, специально предназначенную для подобных целей.

Команда loop выполняет декремент содержимого регистра CX (ECX), и если оно не равно нулю, то осуществляется переход на указанную метку вперед или назад в диапазоне от −128 до +127 байт. Содержимое регистра CX (ECX) рассматривается как целое число без знака. Перед использованием команды loop в регистр CX (ECX) нужно поместить счетчик итераций. Команда loop является последней в цикле и анализирует содержимое счетчика: как только оно становится равным нулю, происходит выход из цикла.

Следующий пример демонстрирует в общих чертах методику использования команды loop:

```
. . .
.data
  counter  DW 5
.code
  . . .
  xor   AX, AX
  mov   CX, counter  ; счетчик итераций -> CX
 next:
  inc   AX           ; инкремент регистра AX
  loop  next         ; следующая итерация
 . . .
```

После окончания цикла регистр AX будет содержать значение 5. Команду loop можно представить ее функциональным аналогом, состоящим из других команд, как показано в этом примере:

```
. . .
.data
  counter  DW 5
.code
  . . .
  xor   AX, AX
  mov   CX, counter  ; счетчик итераций -> CX
 next:
  inc   AX           ; инкремент регистра AX
  dec   CX           ; декремент регистра CX
  jcxz   skip        ; если CX = 0, выйти из цикла
  jmp    next        ; следующая итерация
 . . .
skip:
 . . .
```

Если вместо команды jcxz в этом фрагменте кода применить jz, то исходный текст будет выглядеть так:

```
 . . .
.data
  counter  DW 5
.code
```

```
...
 xor  AX, AX
 mov  CX, counter  ; счетчик итераций -> CX
next:
 inc  AX           ; инкремент регистра AX
 dec  CX           ; декремент регистра CX
 jnz  next         ; если CX = 0, выйти из цикла,
                   ; иначе следующая итерация
...
```

Модификациями команды loop являются команды loope/loopz и loopne/loopnz. Рассмотрим вначале команду loope/loopz. Обозначения loope и loopz представляют собой синонимы и относятся к одной и той же команде. Эта команда обладает дополнительными возможностями по обработке циклов. Она выполняет декремент содержимого регистра CX (ECX), и если оно не равно 0 и флаг ZF установлен в 1, то выполняется переход на указанную метку вперед или назад.

Рассмотрим пример использования команды loope. Это простое 16-разрядное приложение, которое выводит на экран дисплея строку без начальных пробелов (листинг 5.3).

**Листинг 5.3.** Вывод строки без начальных пробелов на экран

```
.model small
.data
 sl   DB "     String with leading blanks !$"
 len  EQU  $-sl
 msg  DB "Blank string!$"
.code
 start:
  mov   AX, @data
  mov   DS, AX
  lea   SI, sl              ; адрес строки -> SI
  dec   SI                  ; декремент адреса для организации цикла
  mov   CX, len             ; размер строки -> CX
  mov   AL, ' '             ; шаблон для сравнения -> AL
next:
  inc   SI                  ; переход к адресу следующего элемента
  cmp   byte ptr  [SI], AL  ; сравнить элемент строки с пробелом
  loope next                ; повторять, пока не будет обнаружен символ,
                            ; отличный от пробела,
                            ; либо не будет достигнут
                            ; конец строки
  cmp   CX, 0               ; был достигнут конец строки?
  je    fail                ; да, строка состоит из пробелов,
                            ; вывести соответствующее сообщение
  mov   DX, SI              ; нет, в строке есть другие символы,
                            ; поместить адрес первого символа,
                            ; отличного от пробела, в регистр DX
show:
  mov   AH, 9h              ; отобразить сообщения
  int   21h
  mov   AH, 1h
  int   21h
```

продолжение

**Листинг 5.3** (продолжение)

```
   mov    AX, 4C00h
   int    21h
fail:
   lea    DX, msg
   jmp    show
   end    start
   end
```

Перейдем к анализу команды loopne/loopnz. Отличие этой команды от loope/loopz состоит в том, что цикл выполняется, пока выполняется условие ZF = 0. Обозначения loopne и loopnz являются синонимами и относятся к одной и той же команде. Пример использования команды loopne приводится в листинге 5.4. Как и в предыдущем примере, это 16-разрядное приложение, которое выводит на экран дисплея часть строки, следующей за символом + (то есть String 2).

**Листинг 5.4.** Вывод на экран части строки после символа +

```
.model small
.data
  s1    DB  "String 1+String 2$"
  len   EQU $-s1
  msg   DB  "Char + not found!$"
.code
  start:
   mov    AX, @data
   mov    DS, AX
   lea    SI, s1
   dec    SI
   mov    CX, len
   mov    AL, '+'
next:
   inc    SI
   cmp    byte ptr  [SI], AL
   loopne next
   cmp    CX, 0
   je     fail
   mov    DX, SI
show:
   mov    AH, 9h
   int    21h
   mov    AH, 1h
   int    21h
   mov    AX, 4C00h
   int    21h
fail:
   lea    DX, msg
   jmp    show
   end    start
   end
```

Как видно из примеров, команда loop и ее модификации очень удобны для организации вычислительных алгоритмов, поскольку избавляют программиста от необходимости дополнительного кодирования и проверки условий. Последняя

команда, которую мы рассмотрим, является модификацией обычной команды loop, но предназначена она для работы с двойными словами. Такая команда очень полезна при разработке 32-разрядных приложений, которые в большинстве случаев оперируют двойными словами.

Эта команда обозначается как loopd, и основное ее отличие от команды loop состоит в том, что в качестве счетчика цикла используется регистр ECX, содержимое которого в конце каждой итерации уменьшается на 4. Напомню, что двойное слово занимает в оперативной памяти 4 байта, а счетчик ECX содержит размер области памяти в байтах. Адрес следующего обрабатываемого элемента отстоит от текущего на 4 в сторону увеличения или уменьшения, в зависимости от алгоритма задачи. Команда loopd оперирует с теми же флагами, что и loop. Должен заметить, что команда loopd не включена в ранние модели процессоров Intel.

В качестве примера выполнения цикла с использованием команды loopd приведу 32-разрядную процедуру, выполняющую поиск в массиве целых чисел первого элемента, меньшего –100. В случае удачного поиска процедура возвращает значение этого элемента в регистре EAX, в случае неудачи — 0 в этом же регистре. Исходный текст процедуры (она называется _loopd_ex) приведен в листинге 5.5.

**Листинг 5.5.** Поиск первого элемента массива, меньшего –100

```
.586
.model flat
option casemap: none
.data
 al  DD 312, -45, 91, -16, -377  ; сканируемый массив
 len EQU $-al                    ; размер массива в байтах
.code
_loopd_ex proc
  mov   ECX, len                 ; размер массива в байтах -> ECX
  shr   ECX, 2                   ; преобразовать размер в двойные слова
  lea   ESI, al                  ; адрес первого элемента -> ESI
  mov   EAX, -100                ; шаблон для сравнения -> EAX
next:
  cmp   EAX, [ESI]               ; сравнить элемент массива
                                 ; с содержимым регистра EAX
  jge   found                    ; число в массиве меньше -100,
                                 ; закончить программу
  add   ESI, 4                   ; число больше -100, перейти
                                 ; к следующему элементу массива
  loopd next                     ; следующая итерация
  jmp   not_found                ; массив проверен, чисел меньше -100 нет
found:
  mov   EAX, [ESI]               ; значение элемента массива -> EAX
  jmp   exit                     ; выйти из процедуры
not_found:
  mov   EAX, 0                   ; при неудачном поиске в регистр EAX
                                 ; помещается 0
exit:
  ret
 _loopd_ex endp
 end
```

Исходный текст процедуры несложен и в дополнительных объяснениях не нуждается, нужно лишь помнить, что процедура оперирует двойными словами, и указывать следующий адрес на 4 больше предыдущего.

## 5.4. Оптимизация кода в процессорах Intel Pentium

До сих пор мы рассматривали различные варианты реализации условных переходов и ветвлений в программах безотносительно к тому, быстро или медленно будет работать тот или иной фрагмент кода. В большинстве случаев программист обычно не задумывается над производительностью работы приложения, учитывая то, что современные аппаратные средства обеспечивают довольно приличный уровень быстродействия и без специальных усилий со стороны разработчика программного обеспечения.

Однако в целом ряде случаев производительность программы выходит на первое место, а система команд процессоров Intel Pentium, особенно для последних моделей, позволяет ее повысить. В этом разделе мы рассмотрим некоторые вопросы, связанные с повышением эффективности программ.

Быстродействие программы в значительной степени определяется алгоритмом вычислений, а также количеством ветвлений и переходов. Кроме этого, большое влияние на производительность программы оказывают характер ветвлений и организация циклических вычислений там, где они необходимы.

В идеале высокопроизводительное приложение должно состоять из линейно выполняющегося программного кода, без ветвлений и переходов. В этом случае процессор Intel Pentium мог бы обеспечить наибольшее быстродействие программы, поскольку не понадобился бы механизм прогнозирования ветвлений, отнимающий приличную часть процессорного времени в цикле выполнения команды.

Поскольку избежать ветвлений и переходов в программах вряд ли когда-нибудь удастся, то можно, по крайней мере, уменьшить их количество или оптимизировать сами ветвления. Разработчики фирмы Intel включили в систему команд процессоров Pentium ряд новых команд, предназначенных для оптимизации переходов в программах. Лучше всего показать работу таких команд на примерах и одновременно изучить их синтаксис.

Для повышения производительности программ фирма Intel включила в новые поколения процессоров, начиная с Pentium II, ряд команд, позволяющих эффективно управлять ветвлениями программы. К таким командам относятся команды set*CC*, cmov*CC* и fcmov*CC*, где *CC* — одно из условий (e, ne, le и т. д.).

Остановимся на синтаксисе этих команд и начнем с команд set*CC*. Вот их формат:

```
setCC reg8
setCC mem8
```

Здесь set*CC* — одна из следующих команд: sete/setz, setl/setnge и т. д., а *reg8*/*mem8* — единственный операнд команды, представляющий собой 8-разрядный регистр, например AL, AH, BL и т. д., или байт памяти. Если заданное в команде условие выполнено, то в операнд помещается значение 1, если ложно — 0. Коман-

ды set*CC* анализируют соответствующие флаги, установленные предыдущими ассемблерными инструкциями.

Проиллюстрируем сказанное примером:

```
cmp AL, 0
sete BL
```

Если после выполнения команды cmp обнаружено равенство нулю содержимого регистра AL, то флаг ZF будет установлен в 1. Следующая команда sete анализирует состояние этого флага и помещает в регистр BL значение 1. Если бы в AL содержалось число, отличное от нуля, то в регистр BL было бы записано значение 0.

Перечень команд set*CC* приведен в табл. 5.4.

**Таблица 5.4.** Команды setCC

| Мнемоника | Описание | Проверяемые флаги |
|---|---|---|
| SETAE/SETNB | Установить, если выше или равно/не ниже | CF |
| SETE/SETZ | Установить, если равно/нуль | ZF |
| SETNE/SETNZ | Установить, если не равно/не нуль | ZF |
| SETB/SETNAE | Установить, если ниже/не выше или равно | CF |
| SETBE/SETNA | Установить, если ниже или равно/не выше | CF, ZF |
| SETL/SETNGE | Установить, если меньше/не больше или равно | SF, OF |
| SETGE/SETNL | Установить, если больше или равно/не меньше | SF, OF |
| SETG/SETNLE | Установить, если больше/не меньше или равно | ZF, SF, OF |
| SETS | Установить, если SF = 1 | SF |
| SETNS | Установить, если SF = 0 | SF |
| SETC | Установить, если CF = 1 | CF |
| SETNC | Установить, если CF = 0 | CF |
| SETO | Установить, если OF = 1 | OF |
| SETNO | Установить, если OF = 0 | OF |
| SETP/SETPE | Установить, если PF = 1 | PF |
| SETNP/SETPO | Установить, если PF = 0 | PF |

Команды set*CC* очень удобны при организации вычислений по условию. При этом можно избавиться от ненужных команд переходов, что дает выигрыш в быстродействии. Рассмотрим следующий пример. Пусть в массиве целых чисел требуется найти первое число, лежащее между 50 и 100. Эту задачу можно решить с помощью процедуры find_num, исходный текст которой показан в листинге 5.6.

В этой процедуре просматривается массив целых чисел a1, адрес которого находится в регистре ESI. Для поиска нужного элемента используется обычный алгоритм, в котором каждый элемент массива проверяется дважды: является ли он меньшим или равным числу 100 (команды cmp dword ptr [ESI], 100 и jle next1), а также большим или равным 50 (команды cmp dword ptr [ESI], 50 и jge found). Применение нескольких команд set*CC* позволяет уменьшить число ветвлений программы.

**Листинг 5.6.** Поиск первого элемента массива, находящегося в диапазоне 50–100

```
.686
.model flat
option casemap: none
.data
  a1   DD 34, -53, 88, 13, 67
  len  EQU $-a1
.code
 find_num  proc
  lea   ESI, a1                 ; адрес массива -> ESI
  mov   ECX, len                ; размер массива в байтах -> ECX
  shr   ECX, 2                  ; преобразовать в количество двойных слов
next:
  cmp   dword ptr [ESI], 100    ; элемент массива меньше или равен 100?
  jle   next1                   ; да, выполним следующую проверку
  jmp   next_addr               ; число больше 100, перейти
                                ; к следующему адресу
next1:
  cmp   dword ptr [ESI], 50     ; элемент массива больше или равен 50?
  jge   found                   ; да, элемент обнаружен, поместить его
                                ; в регистр EAX и выйти из процедуры
next_addr:                      ; перейти к следующему элементу массива
  add   ESI, 4
  dec   ECX                     ; декремент счетчика
  jnz   next                    ; если содержимое ECX не равно 0,
                                ; перейти к следующей итерации
  mov   EAX, 0                  ; цикл завершен, требуемый элемент
                                ; отсутствует, помещаем в EAX значение 0
  jmp   exit
found:
  mov   EAX, [ESI]              ; найденный элемент -> EAX
exit:
  ret
find_num endp
end
```

В листинге 5.7 представлен модифицированный вариант этой же процедуры, в которой в той или иной форме используются команды set*CC*.

**Листинг 5.7.** Модифицированный с использованием команд setCC вариант листинга 5.6

```
.686
.model flat
option casemap: none
.data
  a1   DD 34, -53, 88, 13, 67
  len  EQU $-a1
  g50  DB ?                     ; вспомогательные переменные
  l100 DB ?
.code
 find_num  proc
   lea   ESI, a1                ; адрес массива -> ESI
   mov   ECX, len               ; размер массива в байтах -> ECX
   shr   ECX, 2                 ; преобразовать в количество двойных слов
```

```
next:
   cmp   dword ptr [ESI], 50   ; элемент массива больше или равен 50?
   setge g50                   ; если да, установить переменную g50 в 1,
                               ; иначе установить g50 в 0
   cmp   dword ptr [ESI], 100  ; элемент массива меньше или равен 100?
   setle l100                  ; если да, установить переменную l100 в 1,
                               ; иначе установить l100 в 0
   mov   AL, g50               ; сравнить g50 и l100
   cmp   AL, l100
   je    found                 ; если переменные равны, элемент обнаружен
                               ; и поиск заканчивается
   add   ESI, 4                ; нет, g50 не равен l100, продолжить поиск
   dec   ECX
   jnz   next
   mov   EAX, 0                ; цикл закончен, нужный элемент не найден,
                               ; помещаем в регистр EAX значение 0
   jmp   exit
 found:
   mov   EAX, [ESI]            ; значение обнаруженного элемента -> EAX
 exit:
   ret
 find_num endp
 end
```

В исходном тексте изменения выделены жирным шрифтом. Смысл изменений достаточно очевиден, замечу лишь, что нам удалось избавиться от двух команд условных переходов и сделать программный код более линейным. При указанных значениях элементов массива по завершении процедуры регистр EAX будет содержать число 88.

Следующая группа команд, которую мы рассмотрим, включает команды cmov*CC*. Формат этой команды выглядит так:

```
cmovCC src, dst
```

Здесь *CC* — одно из условий (e, ne, nz, le и т. д.), src может быть 16- или 32-разрядным регистром, а *dst* — 16- или 32-разрядным регистром или ячейкой памяти. Команда проверяет условие и, если оно выполняется, копирует содержимое *dst* в *src*. Если условие не выполняется, операнд *src* остается без изменений. Небольшой пример поможет лучше понять способ использования команд cmov*CC*:

```
 .data
   op1 DW ?
.code
. . .
cmp    AX, op1
cmovge AX, op1
. . .
```

Если содержимое регистра AX больше или равно переменной op1, то op1 копируется в AX. Если же содержимое AX меньше op1, то оба операнда остаются неизменными.

Команда cmov*CC* весьма полезна при разработке быстрых алгоритмов и оптимизации ветвлений. Перед применением команды cmov*CC* необходимо проверить, поддерживается ли она процессором, что легко сделать с помощью команды cpuid.

Рассмотрим еще один пример программы, в которой присутствуют команды условных переходов, и попробуем ее усовершенствовать. Пусть требуется определить большее из двух целых чисел. Если использовать обычные команды ассемблера, то этот алгоритм можно реализовать с помощью следующей последовательности инструкций:

```
. . .
.data
  num1 DD 12
  num2 DD 11
.code
. . .
  clc
  mov EAX, num1
  mov EDX, num2
  cmp EAX, EDX
  jg  num1_g_num2
  mov EBX, EDX
  jmp exit
num1_g_num2:
  mov EBX, EAX
exit:
. . .
```

Этот код сравнивает два целых числа — num1 и num2, помещая большее из них в регистр EBX. Здесь присутствует команда условного перехода jg, выполняющая переход на другую ветвь программного кода, если num1 больше num2. Для модификации программного кода воспользуемся командой cmovl. Новый вариант исходного текста программы выглядит так:

```
. . .
.data
  num1 DD 12
  num2 DD 11
.code
  . . .
 mov   EAX, num1
 mov   EDX, num2
 cmp   EAX, EDX
 cmovl EAX, EDX
 mov   EBX, EAX
  . . .
```

Проанализируем программный код. В регистр EAX помещается первое число (num1), а в EDX — второе (num2). После выполнения показанной ниже команды сравнения будут установлены соответствующие флаги:

```
cmp EAX, EDX
```

Следующая команда помещает в регистр EAX содержимое EDX, если число в EDX больше числа в EAX, и оставляет содержимое EAX без изменения, если число в EAX больше числа в EDX:

```
cmovl EAX, EDX
```

Наконец, содержимое регистра EAX помещается в регистр EBX. Из этого фрагмента видно, что ветвлений и переходов нет. Перед использованием команды

cmov*CC* необходимо проверить, поддерживается ли она данным типом процессора. Такую проверку можно выполнить с помощью команды cpuid.

Рассмотрим еще один пример. Пусть требуется найти модуль (абсолютное значение) числа. Используя обычную команду условного перехода jge, можно сделать это с помощью следующего программного кода:

```
.data
  num1  DD -18
.code
  . . .
  mov   EAX, num1
  cmp   EAX, 0
  jge   exit
  neg   EAX
exit:
  . . .
```

Как видно из исходного текста, после команды cmp программный код разветвляется. Этого можно легко избежать, если использовать команду cmovl. Более быстродействующий код выглядит так:

```
 . . .
.data
  num1  DD 18
.code
 . . .
 mov   EAX, num1
 mov   EDX, EAX
 neg   EDX
 cmp   EAX, 0
 cmovl EAX, EDX
 . . .
```

В нашем последнем примере представлен окончательный вариант процедуры find_num, в которой используются команды set*CC* и cmov*CC* (листинг 5.8).

Ранее мы уже достаточно подробно анализировали исходный текст процедуры find_num, поэтому остановимся лишь на последних изменениях (они выделены жирным шрифтом). Как видно из листинга, в случае равенства переменных g50 и l100 команда cmove EAX, [ESI] копирует искомое значение в регистр EAX. Следующая инструкция je exit передает управление либо на выход процедуры (флаг ZF = 1), либо следующей команде (в данном случае add ESI, 4). Команда cmove EAX, [ESI] не влияет на состояние флагов, поэтому инструкция je exit, фактически, анализирует флаги, установленные командой cmp AL, l100. Как видите, в модифицированном варианте процедуры осталось всего две команды условных переходов.

Более-менее сложная программа, помимо условных и безусловных переходов, может содержать один или несколько вычислительных циклов. Вычисления или другие операции, выполняющиеся циклически, могут повторяться от нескольких десятков до сотен миллионов раз, создавая ощутимую нагрузку на память и процессор. Правильная организация циклов помогает не только повысить производительность программы в целом, но и снизить потребление ресурсов. Сейчас мы проанализируем, каким образом можно повысить производительность программ или процедуры на ассемблере за счет оптимизации циклических вычислений.

**Листинг 5.8.** Поиск элемента массива, находящегося в диапазоне 50–100 (окончательный вариант)

```
.686
.model flat
option casemap: none
.data
  a1   DD 34, -93, 95, 13, 7, 1
  len  EQU $-a1
  g50  DB ?
  l100 DB ?
.code
find_num  proc
   lea   ESI, a1
   mov   ECX, len
   shr   ECX, 2
next:
   cmp   dword ptr [ESI], 50
   setge g50
   cmp   dword ptr [ESI], 100
   setle l100
   mov   AL, g50
   cmp   AL, l100
   cmove EAX, [ESI]
   je    exit
   add   ESI, 4
   dec   ECX
   jnz   next
   mov   EAX, 0
exit:
   ret
find_num endp
end
```

Вначале остановимся на теоретическом аспекте проблемы. В наиболее общем виде цикл представляет собой последовательность команд, начинающихся с определенной метки и возвращающихся на нее после выполнения этой последовательности. В псевдокодах ассемблера цикл можно представить так:

```
метка:
     инструкции ассемблера
     jmp метка
```

Например, следующая последовательность команд представляет собой простой цикл:

```
      xor EBX, EBX
L1:
     инструкции ассемблера
     inc EBX
     cmp EBX, 100000
     je exit
     jmp L1
exit:
```

В этом цикле выполняется увеличение содержимого регистра EBX от 0 до 100 000, и при достижении этого значения передается управление на метку exit.

Проанализируем этот фрагмент кода с точки зрения быстродействия. Его нельзя назвать оптимальным, поскольку для анализа условия выхода из цикла и самого выхода из цикла используется несколько команд переходов. Для цикла с фиксированным числом итераций проблему оптимизации решить несложно, как показано в следующем фрагменте кода:

```
 . . . .
   mov EDX, 100000
 L1:
   . . .
   инструкции ассенблера
   . . .
   dec EDX
   jnz L1
 exit:
   . . .
```

Как видно из этого листинга, значение счетчика цикла помещается в регистр EDX. Этот фрагмент более эффективен, чем предыдущий. Команда декремента устанавливает флаг ZF, когда счетчик становится равным 0, что приводит к выходу из цикла, иначе цикл продолжается. Этот фрагмент кода требует меньшего количества команд и будет выполняться быстрее.

Еще один способ повышения быстродействия циклических вычислений состоит в том, чтобы по возможности избавиться от ветвлений и условных переходов внутри самого цикла. Рассмотрим пример (листинг 5.9). Пусть в массиве целых чисел необходимо заменить все отрицательные числа нулями. Это можно сделать с помощью 32-разрядной процедуры на ассемблере (назовем ее _set0).

**Листинг 5.9.** Замена отрицательных элементов массива целых чисел нулевыми значениями

```
.686
.model flat
option casemap:none
.data
  iarray  DD -73, 931, -89, 92, -5, 67, 30
  len     EQU $-iarray
.code
 _set0 proc
   lea   ESI, iarray          ; адрес массива -> ESI
   mov   EDX, len             ; размер массива (в байтах) -> EDX
   shr   EDX, 2               ; преобразовать в количество двойных слов
next:
   cmp   dword ptr [ESI], 0   ; сравнить элемент массива с нулем
   jge   no_change            ; если больше нуля, оставить без изменения
   mov   dword ptr [ESI], 0   ; если меньше нуля, заменить на 0
no_change:
   add   ESI, 4               ; перейти к следующему элементу
   dec   EDX                  ; уменьшить счетчик цикла на 1
   jnz   next                 ; переход к следующей итерации
   lea   EAX, iarray          ; адрес массива -> EAX
   ret
 _set0 endp
 end
```

В этой процедуре выполняется основной цикл, операторы которого находятся между меткой next и инструкцией jnz next и образуют тело цикла. В теле цикла присутствует команда jge no_change, выполняющая ветвление в зависимости от результата сравнения. Подобные изменения в последовательности выполнения отрицательно сказываются на быстродействии, поэтому попробуем избавиться от лишнего перехода. Сделать это можно с помощью команды setge. Посмотрим, как будет выглядеть модифицированный вариант процедуры (листинг 5.10).

**Листинг 5.10.** Модифицированный вариант листинга 5.9, в котором используется команда setge

```
.686
.model flat
option casemap:none
.data
  iarray  DD 273, 417, -31, -92, 5, -67, 360
  len     EQU $-iarray
.code
 _set0 proc
   push  EBX
   lea   ESI, iarray
   mov   EDX, len
   shr   EDX, 2
next:
   xor   EBX, EBX
   cmp   dword ptr [ESI], 0
   setge BL
   imul  EBX, dword ptr [ESI]
   mov   dword ptr [ESI], EBX
   add   ESI, 4
   dec   EDX
   jnz   next
   lea   EAX, iarray
   pop   EBX
   ret
 _set0 endp
 end
```

Хочу отметить, что даже хорошо оптимизированный цикл иногда не работает так быстро, как ожидает разработчик. Для дальнейшего повышения эффективности прибегают к так называемому разворачиванию (unrolling) цикла. Этот термин означает на самом деле, что цикл должен выполнять больше действий в одной итерации для уменьшения количества итераций. Это дает неплохой эффект, и сейчас мы рассмотрим два фрагмента программного кода, в которых имеет место разворачивание циклов.

В качестве исходного (неоптимизированного) фрагмента кода возьмем, например, копирование данных размером в двойное слово из одного буфера памяти (обозначим его как src) в другой (dst). Исходный текст представлен в листинге 5.11.

**Листинг 5.11.** Копирование двойных слов без оптимизации

```
. . .
.data
  src  DD 345, -65, 12, 99, 369, 267
  len  EQU $-src
  dst  DD 6 DUP (?)
.code
. . .
  mov ESI, src       ; адрес источника src-> ESI
  mov EDI, dst       ; адрес приемника dst -> EDI
  mov ECX, len       ; значение счетчика байтов -> ECX
  shr ECX, 2         ; перевести значение счетчика
                     ; в двойные слова
L1:
  mov EAX, [ESI]
  add ESI, 4
  mov [EDI], EAX
  add EDI, 4
  dec ECX
  jnz L1
. . .
```

Для разворачивания цикла выполним одновременно копирование двух двойных слов. Исходный текст оптимизированного фрагмента кода показан в листинге 5.12 (изменения выделены жирным шрифтом).

**Листинг 5.12.** Модифицированный код листинга 5.11

```
. . .
.data
  src  DD 345, -65, 12, 99, 369, 267
  len  EQU $-src
  dst  DD 6 DUP (?)
.code
. . .
  mov ESI, src         ; адрес источника src -> ESI
  mov EDI, dst         ; адрес приемника dst -> EDI
  mov ECX, len         ; значение счетчика байтов -> ECX
  shr ECX, 3           ; перевести значение счетчика
                       ; в учетверенные слова (два
                       ; двойных слова)
L1:
  mov EAX, [ESI]       ; сохраняем первое двойное слово из
                       ; пары в регистре EAX
  mov EBX, [ESI + 4]   ; сохраняем второе двойное слово в регистр EBX
  mov [EDI], EAX       ; помещаем первое двойное слово в регистр EDI
  mov [EDI + 4], EBX   ; записываем второе двойное слово по адресу
                       ; в регистре EDI на 4 больше предыдущего
  add ESI, 8           ; продвигаем адреса источника и приемника так, чтобы
  add EDI, 8           ; они указывали на следующее двойное слово
  dec ECX
  jnz label            ; переход к следующей итерации или
                       ; выход из цикла
  . . .
```

Разворачивание позволяет наполовину скомпенсировать снижение производительности программы, в которой используется такой цикл. Если оперировать не двумя, а четырьмя двойными словами, то можно развернуть цикл далее.

Приведу еще один пример разворачивания циклов. Пусть имеется массив из 10 целых чисел и требуется присвоить элементам массива с четными номерами значение 0, а элементам с нечетными номерами значение 1. Если особо не задумываться над качеством программы, то можно быстро написать фрагмент программного кода, представленный в листинге 5.13.

**Листинг 5.13.** Обработка четных и нечетных элементов целочисленного массива

```
. . .
.data
  iarray DD 10 dup (0)
  len  EQU $-iarray
.code
. . .
  mov  ECX, len            ; число элементов массива (в байтах) -> ECX
  lea  ESI, i1             ; адрес первого элемента массива -> ESI
  mov  EBX, 2              ; помещаем делитель 2 в регистр EBX для
                           ; определения, четный или нечетный элемент
next:
  mov  EAX, ECX            ; счетчик элементов -> EAX
  div  EBX                 ; определяем, четный или нечетный
                           ; порядковый номер у элемента массива
  cmp  EDX, 0
  jne  store_1             ; если нечетный, присваиваем элементу
                           ; значение 1
  mov  DWORD PTR [ESI], 0  ; если четный, присваиваем элементу значение 0
  jmp  next_addr
store_1:
  mov  DWORD PTR [ESI], 1
next_addr:                 ; адрес следующего элемента массива
  add  ESI, 4
  loop next
  . . .
```

Данный фрагмент программного кода можно оптимизировать, если обрабатывать в каждой итерации два двойных слова вместо одного. Модифицируем предыдущий пример, поместив программный код в процедуру unr_1. Исходный текст измененной программы показан в листинге 5.14.

**Листинг 5.14.** Модифицированный код листинга 5.13 с разворачиванием цикла

```
.686
.model flat
option casemap: none
.data
  iarray DD 10 dup (7)
  len EQU $-iarray
.code
_unr_1 proc
  lea ESI, iarray
  mov EBX, len
```

```
  shr  EBX, 2
  dec  EBX
  xor  EDX, EDX
next:
  mov  DWORD PTR [ESI], 0
  mov  DWORD PTR [ESI+4], 1
  add  EDX, 2
  cmp  EDX, EBX
  jae  exit
  add  ESI, 8
  jmp  next
exit:
  lea  EAX, array
  ret
  _unr_1 endp
  end
```

Как видите, исходный текст этого фрагмента кода претерпел существенные изменения по сравнению с предыдущим примером. Программа стала более компактной; повысилась ее производительность, поскольку мы избавились от команд деления и одновременно уменьшили число итераций в два раза.

В каждой итерации обрабатываются одновременно два элемента массива командами

```
mov  DWORD PTR [ESI], 0
mov  DWORD PTR [ESI+4],1
```

В конце каждой итерации содержимое регистра ESI увеличивается на 8 с помощью команды add ESI,8, указывая на следующую пару элементов. Количество обрабатываемых пар элементов помещается в регистр EBX:

```
mov  EBX, len
shr  EBX, 2
dec  EBX
```

Здесь хочу сделать важное замечание. В нашей процедуре обрабатывается 10 двойных слов, поэтому регистр EBX должен содержать значение 9 для корректной работы цикла. Если количество элементов массива будет нечетным, то необходимо обрабатывать последнее двойное слово вне цикла. Это потребует дополнительных команд, но в целом не окажет существенного влияния на быстродействие процедуры, особенно при больших размерах обрабатываемых массивов. Например, чтобы обработать 1589 двойных слов, объединив каждые два элемента, необходимо выполнить 397 итераций для учетверенных слов и после окончания цикла обработать одно двойное слово. При желании читатели могут самостоятельно разработать подобную процедуру, обрабатывающую произвольное количество двойных слов.

Для организации циклических вычислений очень часто используются команда loop и ее модификации. Соответствующие примеры мы рассматривали ранее в этой главе. Эта команда очень удобна, поскольку избавляет программиста от необходимости постоянно проверять условие окончания цикла. Модификации команды loop, такие, например, как loope и loopne, еще больше упрощают программирование циклов.

Несмотря на очевидные удобства в применении, команда loop имеет средние показатели производительности. Если на первое место выходит скорость выполнения программного кода, то команду loop лучше не использовать, особенно при обработке большого числа элементов строк или массивов. В таких случаях желательно заменить команду loop группой команд. Это замечание касается, в первую очередь, приложений, разрабатываемых для процессоров Intel Pentium, поскольку на более ранних процессорах команда loop работает быстрее своих программных аналогов.

Вот пример замены команды loop эквивалентными ей командами:

```
dest:
    . . .
    dec cx
    jnz dest
    . . .
```

Что же касается команд loope и loopne, то они работают значительно медленнее, чем эквивалентный им код, включающий обычные команды процессоров Intel Pentium. При очень интенсивных вычислениях команды loop*CC* (*CC* = e, ne, z, nz) в программах лучше не использовать. Стандартной эквивалентной замены для таких команд не существует, поскольку в каждом конкретном случае программный код может быть уникальным. Рассмотрим вариант замены команды loope в приведенном ранее примере 16-разрядного приложения (см. листинг 5.3).

Напомню, что программный код примера выводит на экран строку без начальных символов пробела. В листинге 5.15 показан исходный текст модифицированной программы.

**Листинг 5.15.** Модифицированный код листинга 5.3

```
.model small
.data
  s1    DB "     String with leading blanks !$"
  len   EQU   $-s1
  msg   DB "Blank string!$"
.code
  start:
   mov   AX, @data
   mov   DS, AX
   lea   SI, s1
   dec   SI
   mov   CX, len
   mov   AL, ' '
next:
   inc   SI
   cmp   byte ptr  [SI], AL
   jne   $+7
   dec   CX
   jnz   next
   jmp   fail
   mov   DX, SI
show:
   mov   AH, 9h
```

```
  int   21h
  mov   AH, 1h
  int   21h
  mov   AX, 4C00h
  int   21h
fail:
  lea   DX, msg
  jmp   show
  end   start
  end
```

В этой программе команда loope заменена следующим фрагментом кода (выделен жирным шрифтом):

```
  . . .
  jne   $+7
  dec   CX
  jnz   next
  . . .
```

Как работает эта группа команд? На каждой итерации выполняется поиск символа пробела с помощью команды

```
cmp   byte ptr  [SI], AL
```

Предположим, что обнаружен символ, отличный от пробела. В этом случае команда cmp устанавливает флаг ZF в 0. Следующая команда jne $+7 анализирует флаг ZF и передает управление команде, находящейся по адресу со смещением +7 в сегменте программного кода. Это смещение определяется как разность адресов следующей выполняемой команды и текущей. Следующей командой является

```
mov   DX, SI
```

Она загружает адрес оставшейся части строки в регистр DX для вывода на экран. Эта команда отстоит на 7 байт от выполняемой в данный момент команды. Таким образом, команда jne $+7 передает управление по адресу команды

```
mov   DX, SI
```

Если обнаруженный символ является пробелом, то выполняется декремент содержимого регистра CX, и если оно не равно 0, то цикл повторяется. Если строка состоит из одних пробелов, то после окончания цикла управление передается команде

```
jmp   fail
```

Попробуем теперь подобрать аналог программного кода для команды loopne, которая используется в программе, отображающей часть строки после знака + (см. листинг 5.4). Исходный текст модифицированной программы представлен в листинге 5.16.

Исходный текст фрагмента кода, используемого вместо команды loopne, выделен жирным шрифтом. Он очень напоминает программный код из предыдущего примера, с той лишь разницей, что команда jne по смыслу программы заменена командой je, кроме того, изменилась величина смещения (8 вместо 7). Смещение зависит от объема памяти, занимаемого пропускаемыми командами, а в этом

фрагменте вместо dec CX используется для разнообразия команда dec CL, занимающая объем памяти на 1 байт больше.

**Листинг 5.16. Замена команды loopne в программе из листинга 5.4**

```
.model small
.data
  s1    DB "String 1+String 2$"
  len   EQU  $-s1
  msg DB "Char + not found!$"
.code
  start:
   mov   AX, @data
   mov   DS, AX
   lea   SI, s1
   dec   SI
   mov   CL, len
   mov   AL, '+'
next:
   inc   SI
   cmp   byte ptr  [SI], AL
   je    $+8
   dec   CL
   jnz   next
   jmp   fail
   mov   DX, SI
   inc   DX
show:
   mov   AH, 9h
   int   21h
   mov   AH, 1h
   int   21h
   mov   AX, 4C00h
   int   21h
fail:
   lea   DX, msg
   jmp   show
   end   start
   end
```

Помимо рассмотренных простейших вариантов можно разработать и другие способы модификации программного кода с командами loop*CC*. Автор надеется, что материал этой главы окажет помощь в создании новых, более эффективных алгоритмов обработки данных и модификации уже существующих.

# Процедуры на языке ассемблера



В большинстве программ встречаются фрагменты программного кода, которые нужно неоднократно выполнять и, следовательно, повторять одну и ту же последовательность команд. Такие фрагменты программного кода целесообразно выделить из программы, оформив в виде подпрограмм или процедур, и обращаться к ним всякий раз, когда основной программе потребуется их выполнение.

Немного о терминологии. В дальнейшем термины «подпрограмма» и «процедура» будут использоваться как синонимы (процедура является одной из форм реализации подпрограммы). Везде в этой главе и далее будем считать термины «подпрограмма» и «процедура» тождественными и полагать, что оба они представляют группу команд, заключенных между директивами proc и endp. По отношению к подпрограмме, или процедуре, остальную часть программы принято называть основной или вызывающей программой. Эта глава посвящена принципам разработки подпрограмм (процедур) на языке ассемблера.

Подпрограммы могут находиться как в исполняемом файле основной программы, так и в отдельном объектном файле, который включается в файл основной программы при помощи компоновщика. Это означает, что исходный текст подпрограммы может помещаться в файл с расширением ASM и компилироваться автономно в файл объектного модуля, имеющий расширение OBJ.

Существует еще один способ использования автономных подпрограмм, который нередко применяется в 32-разрядных приложениях Windows: можно создать библиотеку динамической компоновки (Dynamic Link Library, DLL), поместив в нее программный код процедуры. В этом случае основная программа сможет определенным образом получить доступ к процедуре, находящейся в DLL. Создание и функционирование DLL тесно связано с архитектурой операционных систем Windows, что само по себе является отдельной темой, поэтому ограничимся рассмотрением классического варианта применения подпрограмм с использованием объектных файлов.

Для функционирования подпрограмм большое значение имеет правильное использование механизма стековых операций, поэтому прежде всего проанализируем принципы выполнения таких операций.

## 6.1. Организация стека

Стек представляет собой специальную область памяти, которая служит для временного хранения данных и адресов. Для адресации стека используются регистры SS:SP (16-разрядные приложения) и SS:ESP (32-разрядные программы). Регистр SP (ESP) называется указателем стека и содержит 16- или 32-разрядный адрес последнего элемента, помещенного в стек. Последнее значение, помещенное в стек, извлекается первым. Подобная структура называется LIFO (Last In, First Out — прибыл последним, обслужен первым). Стек растет к меньшим адресам, то есть последнее значение, поступившее в стек, хранится по наименьшему адресу.

Несмотря на то что память в процессорах x86 имеет байтовую организацию, минимальный размер операнда, которым оперируют команды стековых операций, равен слову (2 байта). По этой причине данные в стеке отстоят друг от друга на величину, кратную двум. Например, при помещении в стек слова значение указателя стека SP (ESP) уменьшается на 2, при помещении двойного слова — на 4 и т. д. При этом младшие байты операндов помещаются в стек по младшим адресам, а старшие байты — по старшим адресам.

Для того чтобы поместить какое-либо значение в стек, нужно использовать команду push. Эта команда в качестве параметра может принимать любой 16- или 32-разрядный регистр либо ячейку памяти. При этом содержимое указателя стека SP (ESP) уменьшается на 2 (для слова) или на 4 (для двойного слова). Команда допускает один из форматов:

```
push reg16/reg32
push mem16/mem32
push segreg
push immed
```

Здесь *reg16/reg32* — один из 16- или 32-разрядных регистров, *mem16/mem32* — переменная в памяти (16 или 32 разряда), *segreg* — один из сегментных регистров (CS, DS, ES), а *immed* — непосредственное значение. Команда push с непосредственным операндом (immed) в процессорах Intel Pentium недопустима.

Существуют специальные модификации команды push. Так, например, для сохранения 16-разрядного регистра флагов процессора в стеке используется команда pushf, а для сохранения 32-разрядного регистра флагов — команда pushfd. Последняя команда присутствует только в процессорах, начиная с 80386. Наконец, существуют специальные форматы команды push, позволяющие сохранить в стеке все регистры процессора:

- pusha — помещает в стек все 16-разрядные регистры (AX, BX, CX, DX, SP, BP, SI, DI);
- pushad — помещает в стек все 32-разрядные регистры (EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI).

Приведу несколько примеров использования команды push и ее модификаций.

Предположим, что в стеке находится единственное значение, равное 7EE3h (рис. 6.1).

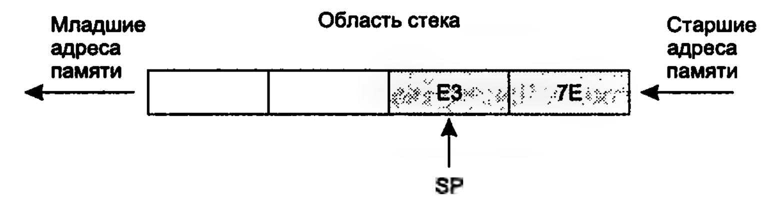



**Рис. 6.1.** Начальное состояние стека

Выполним команды

```
mov  BX, 2CE9h
push BX
```

Команда push в этом фрагменте программного кода копирует содержимое регистра BX в стек, при этом содержимое регистра SP уменьшается на 2 и стек начинает выглядеть так, как показано на рис. 6.2.

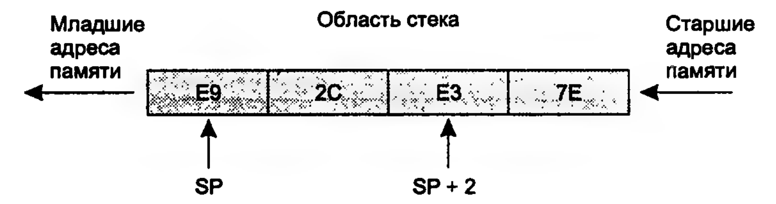



**Рис. 6.2.** Состояние стека после выполнения команды push BX

Напомню, что минимальная размерность данных, которыми оперирует стек, равна 16 бит, поэтому содержимое регистра SP (ESP) не может увеличиться или уменьшиться на 1. Это означает, что нельзя поместить в стек или извлечь из стека данные размером в 1 байт. Указатель стека увеличивается (уменьшается) на 2 или 4 (для слова или двойного слова соответственно). Например, после выполнения следующего фрагмента кода содержимое стека будет таким, как показано на рис. 6.3:

```
mov  EBX, 4FE91A77h
push EBX
```

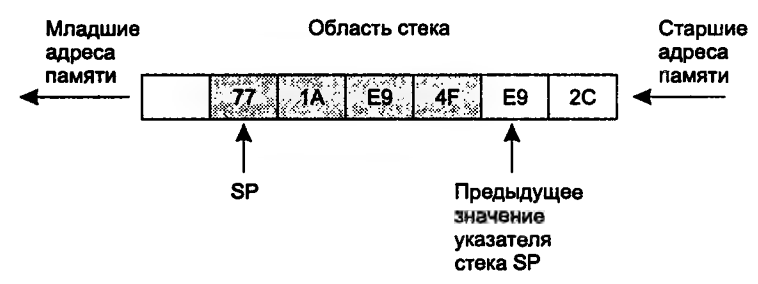



**Рис. 6.3.** Размещение двойного слова в стеке

После этой операции указатель стека уменьшается на 4, поскольку в него помещается двойное слово.

Извлечение данных из стека выполняется с помощью команды pop. При этом из стека извлекается слово (двойное слово) и помещается в указанный операнд. Эта команда в качестве параметра может принимать любой 16- или 32-разрядный регистр или ячейку памяти. При этом содержимое указателя стека SP (ESP) увеличивается на 2 (для слова) или на 4 (для двойного слова).

Команда pop является зеркальной по отношению к push и использует те же типы операндов, что и команда push. Кроме того, для извлечения содержимого регистра флагов из стека имеются команды popf (для 16-разрядного регистра флагов) и popfd (для 32-разрядного). Для того чтобы восстановить все регистры процессора значениями из стека, в систему команд включены инструкции popa (для 16-разрядных регистров) и popad (для 32-разрядных). Например, следующая команда извлекает данные, помещенные в стек в предыдущем примере, и запоминает их в регистре EDX:

```
pop EDX
```

После выполнения этой команды регистр EDX будет содержать значение 4FE91A77h, а указатель стека уменьшится на 4 (рис. 6.4).

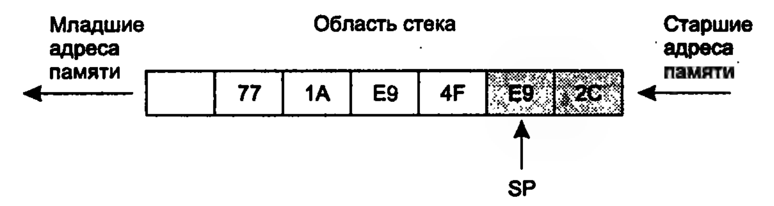



**Рис. 6.4.** Содержимое стека после выполнения команды pop EDX

Как видно из предыдущих примеров, стек может обеспечивать временное хранение данных. Кроме того, с помощью команд push и pop можно организовать обмен данными между регистрами и памятью, причем операнды могут иметь разный размер. В следующих примерах показана техника использования стека в различных операциях:

```
mov  EAX, 11223344h
push EAX
pop  BX
pop  CX
```

Здесь команда push EAX помещает в стек двойное слово 11223344h. После выполнения команды pop BX из стека извлекается младшее слово, равное 3344h, и помещается в регистр BX. Указатель стека ESP при этом уменьшается на 2. Следующая команда pop CX извлекает из стека старшее слово, равное 1122h, и помещает его в регистр CX. При этом содержимое регистра ESP опять уменьшается на 2.

```
. . .
.data
  op DW 7777h
.code
```

```
. . .
push DS:op
pop  AX
. . .
```

В этом примере в стек помещается значение 16-разрядной переменной op (команда push DS:op), при этом указывается сегмент данных, в котором определена переменная (регистр DS). Указатель стека SP после выполнения этой операции уменьшается на 2. Следующая команда pop AX извлекает содержимое стека в регистр AX и восстанавливает стек, увеличивая значение SP на 2. Таким образом, в регистре AX будет содержаться значение 7777h.

Следующий пример демонстрирует применение операций со стеком в 16-разрядном приложении. Исходный текст программы показан в листинге 6.1.

**Листинг 6.1.** Демонстрация стековых операций (16-разрядная версия)

```
.model small
.data
 num1 DW '91'
 s1   DB  "STRING 1 $"
 s2   DB  "STRING 2 $"
.code
start:
  mov  AX, @data
  mov  DS, AX
  push DS:num1
  lea  SI, s2
  push SI
  lea  DX, s1
  mov  AH, 9h
  int  21h
  pop  DX
  int  21h
  pop  DX
  xchg DH, DL
  mov  AH, 2h
  int  21h
  xchg DH, DL
  int  21h
  mov  AX, 4c00h
  int  21h
  end  start
  end
```

Программа достаточно проста — она выводит на экран значения переменной num1 и символьных строк s1 и s2, причем вначале отображается содержимое строки s1, затем — строки s2 и наконец — значение переменной num1. Сначала в стек помещается значение переменной num1 (команда push DS:num1), затем — адрес строки s2:

```
push DS:num1
lea  SI, s2
push s2
```

После этих операций указатель стека уменьшается на 4, а содержимое стека становится таким, как показано на рис. 6.5.

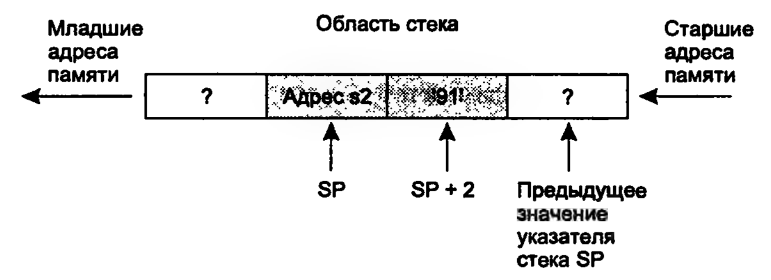



**Рис. 6.5.** Содержимое стека после помещения данных программы

Затем программа выводит на экран строку s1:

```
lea  DX, s1
mov  AH, 9h
int  21h
```

После этого из стека извлекается адрес строки s2 и помещается в регистр DX. Далее строка s2 выводится на экран:

```
pop  DX
int  21h
```

К этому моменту в стеке остается значение переменной num1, а указатель стека SP уменьшается на 2. Следующая команда pop DX извлекает переменную num1 из стека и помещает ее значение в регистр DX, при этом указатель стека еще раз уменьшается на 2. Последующие команды отображают содержимое DX на экране с учетом порядка размещения байтов в регистре:

```
pop  DX
xchg DH, DL
mov  AH, 2h
int  21h
xchg DH, DL
int  21h
```

Хочу сделать замечание: для временного хранения в стеке данных, представленных строками или массивами, используются их адреса или, как их еще называют, указатели. Адрес строки (или массива) одновременно является и адресом ее первого элемента. Например, адрес строки s1 из предыдущего примера совпадает с адресом символа S.

При выполнении операций со стеком вся ответственность за содержимое стека ложится на программиста, поэтому нужно быть очень внимательным. Если какое-либо значение помещается в стек во время работы программы, то оно должно быть извлечено из стека перед ее завершением либо стек должен быть восстановлен каким-то другим способом. Несоблюдение этих требований приводит, как правило, к краху программы. Точно так же суммарный размер операндов, извлеченных из стека, должен быть равным размеру помещенных в него данных.

Хорошо спроектированная программа перед завершением всегда восстанавливает указатель стека к тому значению, которое было перед началом ее выполнения.

Для операций с данными в стеке не обязательно использовать команды push и pop. Вспомним, что стек представляет собой всего лишь область оперативной памяти, поэтому для доступа к данным можно применять обычные команды ассемблера, используя регистровую косвенную адресацию посредством регистра BP (EBP). Для доступа к данным стека необходимо поместить содержимое указателя стека SP (ESP) в регистр BP (EBP), после чего указать смещение данных. Следующий фрагмент программного кода демонстрирует такой подход (листинг 6.2).

**Листинг 6.2.** Доступ к данным в стеке посредством регистра EBP (16-разрядная версия)

```
. . .
.data
  op1 DW 1149h
  op2 DW 0E37h
.code
. . .
mov   AX, @data
mov   DS, AX
push  DS:op1
push  DS:op2
mov   BP, SP
mov   AX, word ptr [BP+2]
mov   BX, word ptr [BP]
. . .
```

Здесь содержимое переменных op1 и op2 помещается в стек, причем значение op1 оказывается по адресу [SP+2], а значение op2 — по адресу [SP] (рис. 6.6).

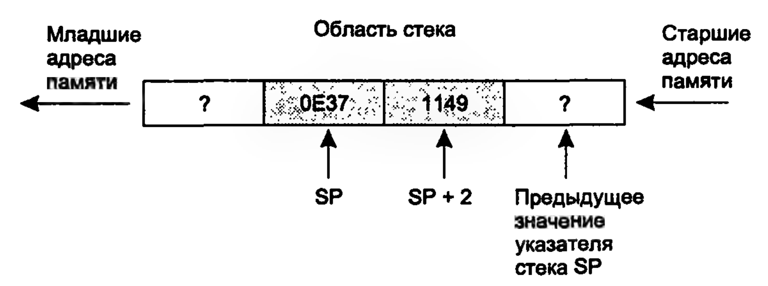



**Рис. 6.6.** Содержимое стека после размещения переменных op1 и op2

Поскольку после выполнения команды mov BP, SP регистр BP содержит значение SP, то значение переменной op1 хранится по адресу [BP+2], а значение op2 — по адресу [BP]. После выполнения последних двух команд данного фрагмента кода регистр AX будет содержать 1149h, а регистр BX — 0E37h.

При разработке 32-разрядных приложений для процессоров Intel Pentium использовать регистр EBP для доступа к данным в стеке не обязательно — можно напрямую работать с указателем стека ESP. Например, с помощью следующего

фрагмента программного кода вычисляется разность операндов op2 и op1, которая затем помещается в регистр EAX:

```
.686
.model flat
option casemap: none
.data
  op1 DD 145
  op2 DD 98
.code
  . . .
  push op1
  push op2
  mov  EAX, dword ptr [ESP]   ; содержимое op1 -> EAX
  sub  EAX, dword ptr [ESP+4] ; op2 - op1 -> EAX
  . . .
```

В последних двух примерах мы не акцентировали внимание на восстановлении указателя стека, хотя в ряде случаев применение обычных команд pop может оказаться неудобным или невозможным. В таких случаях можно воспользоваться еще одним способом восстановления стека — задействовать команду add:

```
add ESP, n
```

Здесь *n* — количество байтов, на которое следует продвинуть указатель стека SP (ESP). Следующий пример демонстрирует восстановление указателя стека после того, как в стек были помещены три двойных слова (12 байт):

```
.code
. . .
push  EAX
push  EBX
push  ECX
. . .
add   ESP, 12
. . .
```

Поскольку команды push помещают в стек 12 байт (три двойных слова), то для восстановления указателя стека следует продвинуть его на это же число в сторону увеличения адресов, что и делается с помощью команды add.

Далее мы проанализируем, как используется стек при выполнении подпрограмм.

# 6.2. Принципы организации подпрограмм

Подпрограмма, в зависимости от выполняемых ею функций, может требовать передачи из вызывающей программы определенных данных, которые принято называть аргументами или параметрами и возвращать в вызывающую программу результат вычислений. Некоторые подпрограммы могут вообще не принимать никаких параметров и не возвращать результат. Чаще всего подпрограмма (процедура) оформляется так, как показано в следующем фрагменте кода:

```
. . .
mov AX, 0
mov BX, 0
```

```
 jmp start
 addl proc  ; точка входа в процедуру addl
  inc AX
  ret       ; возврат в вызывающую программу
 addl endp
 subl proc  ; точка входа в процедуру subl
  dec BX
  ret       ; возврат в вызывающую программу
 subl endp
start:
 call addl  ; вызов процедуры addl
 call subl  ; вызов процедуры subl
 jmp  start
```

Как видно из приведенного фрагмента кода, в начале процедуры (перед первой выполняемой командой) должна находиться директива proc, а после последней выполняемой команды — директива endp. Процедура обязательно должна заканчиваться командой ret. В одном ассемблерном файле с расширением ASM можно размещать несколько процедур.

Точкой входа в процедуру считается директива proc. В директиве proc после имени процедуры не ставится двоеточие, хотя имя считается меткой и указывает на первую команду процедуры. Имя процедуры можно указать в команде перехода, и тогда будет осуществлен переход на первую команду процедуры.

Директива proc может принимать один из двух параметров: near или far. Параметр near указывает на то, что процедура является ближней, а far указывает на то, что процедура дальняя. Если параметр отсутствует, то считается, что процедура имеет тип near (поэтому параметр near обычно и не указывается).

К ближней (near) процедуре можно обращаться только из того сегмента команд, где она объявлена, а к дальней (far) процедуре — из любых сегментов команд, включая тот, где она объявлена. Для 32-разрядных приложений все вызовы процедур считаются ближними.

Следует отметить, что в языке ассемблера имена и метки, описанные в процедуре, должны быть уникальными и не должны совпадать с другими именами в программе. В языке ассемблера имеется возможность создавать вложенные процедуры, то есть процедуры внутри процедур, но особых преимуществ это не дает и используется относительно редко.

Можно обойтись и без явного определения процедуры, пометив первую строку программы некоторой меткой, как проиллюстрировано в следующем фрагменте программного кода:

```
 mov AX, 0
 mov BX, 0
next:
 call addl
 call subl
 jmp  next
addl:       ; подпрограмма, начинающаяся с метки
 inc AX
 ret
subl:       ; подпрограмма, начинающаяся с метки
 dec BX
 ret
```

В этом случае отсутствуют директивы proc и endp и говорят, что подпрограмма (процедура) определяется неявно. Подобные записи процедур используются редко, поскольку значительно затрудняют анализ исходных текстов и отладку программ. Мы не будем применять такое определение процедур, а воспользуемся директивами proc и endp, как было сказано в начале главы. Вызов процедуры выполняется с помощью команды call, которая передает управление процедуре, сохранив в стеке адрес возврата в вызывающую программу. Процедура должна завершаться командой ret, которая извлекает из стека адрес возврата и возвращает управление команде, следующей за командой call.

Рассмотрим более подробно механизмы работы команд call и ret. Особое значение в механизме вызова процедуры и возврата из нее имеет стек. Поскольку в стеке хранится адрес возврата, то процедура, использующая его для хранения промежуточных результатов, должна к моменту выполнения команды ret восстановить стек в том состоянии, в котором он находился перед ее вызовом. В этом случае говорят, что процедура должна восстановить, или очистить, стек.

В момент вызова процедуры команда call помещает в стек адрес команды, следующей непосредственно за call, уменьшая значение указателя стека SP (ESP). Команда ret вызываемой процедуры использует этот адрес для возврата в вызывающую программу, автоматически увеличивая при этом указатель вершины стека.

Типы адресации (near или far) команд ret и call должны соответствовать друг другу. Вызываемая процедура может вызвать с помощью команды call следующую процедуру и т. д., поэтому стек должен иметь достаточный размер для того, чтобы хранить в нем все записываемые данные.

Следует сказать, что команда ret не анализирует состояние или содержимое стека. Она извлекает из вершины стека слово или двойное слово, в зависимости от типа адресации, полагая, что это адрес возврата, по которому передается управление. Если к моменту выполнения команды ret указатель стека окажется смещенным в ту или иную сторону, содержимое вершины стека может представлять все что угодно, поэтому передача управления по этому адресу приведет к краху программы.

Команда call может иметь один из перечисленных ниже форматов вызова:

- прямой ближний (в пределах текущего программного сегмента);
- прямой дальний (вызов процедуры, расположенной в другом программном сегменте);
- косвенный ближний (в пределах текущего программного сегмента с использованием переменной, содержащей адрес перехода);
- косвенный дальний (вызов процедуры, расположенной в другом программном сегменте, с использованием переменной, содержащей адрес перехода).

Тип адресации при вызове процедуры зависит от используемой модели памяти. Директива .model автоматически устанавливает атрибут near или far для

вызываемых процедур, при этом модели tiny, small и compact устанавливают атрибут near, а модели medium, large и huge — атрибут far. Ассемблер генерирует far-вызовы для моделей medium, large и huge автоматически. Для 32-разрядных приложений, использующих модель flat, все вызовы процедур считаются ближними (near).

Проанализируем более подробно форматы вызовов команды call. Если используется прямой ближний вызов, то команда call помещает в стек относительный адрес точки возврата в текущем программном сегменте и модифицирует указатель адресов команд EIP так, чтобы в нем содержался относительный адрес точки перехода в том же программном сегменте.

Требуемая для вычисления этого адреса величина смещения от точки возврата до точки перехода содержится в коде самой команды, занимающем 3 байта (код операции E8h плюс смещение к точке перехода).

Команда call прямого дальнего вызова помещает в стек два слова: вначале сегментный адрес текущего программного сегмента, затем относительный адрес точки возврата в этом программном сегменте. После этого выполняется модификация регистров EIP и CS: в EIP помещается относительный адрес точки перехода в том сегменте, куда осуществляется переход, а в CS — селектор адреса для этого сегмента.

Оба эти значения извлекаются из кода команды, занимающего 5 байт (код операции 9Ah, эффективный адрес вызываемой процедуры и селектор сегмента). Для указания прямого дальнего вызова используется директива far ptr, которая говорит компилятору и компоновщику, что вызов является дальним.

В листинге 6.3 показан фрагмент программного кода, демонстрирующий дальний вызов процедуры.

**Листинг 6.3.** Демонстрация дальних вызовов процедур (16-разрядная версия)

```
.model large
data segment
  s1   DB 0dh, 0ah, "Direct far call of subr1 demo !$"
  s2   DB 0dh, 0ah, "Direct far call of subr2 demo !$"
data ends
code1 segment
assume CS:code1
 main proc              ; точка входа в основную программу
  mov  AX, @data
  mov  DS, AX
  call far ptr subr1    ; дальний вызов подпрограммы subr1
  call far ptr subr2    ; дальний вызов подпрограммы subr2
                        ; код команды call в обоих
                        ; случаях: 9A <смещение> <сегмент>
  mov  AH, 1h
  int  21h
  mov  Ax, 4C00h
  int  21h
 main  endp
code1  ends
```

продолжение

**Листинг 6.3.** (продолжение)

```
. . .
code2 segment
assume CS:code2
subr1 proc far        ; объявление дальней подпрограммы subr1
   lea DX, s1
   mov AH, 9h
   int 21h
   ret                ; команда ret имеет код 0CBh (возврат из дальней
                      ; подпрограммы)
subr1 endp
subr2 proc far        ; объявление дальней подпрограммы subr2
   lea DX, s2
   mov AH, 9h
   int 21h
   ret                ; команда ret имеет код 0CBh (возврат из дальней
                      ; подпрограммы)
subr2 endp
code2 ends
. . .
```

Процедуры subr1 и subr2 находятся в другом сегменте команд той же программы и при вызове выводят на экран строки s1 и s2. При выполнении команды call процессор помещает в стек сначала сегментный адрес вызывающей программы, а затем относительный адрес возврата, как показано на рис. 6.7.

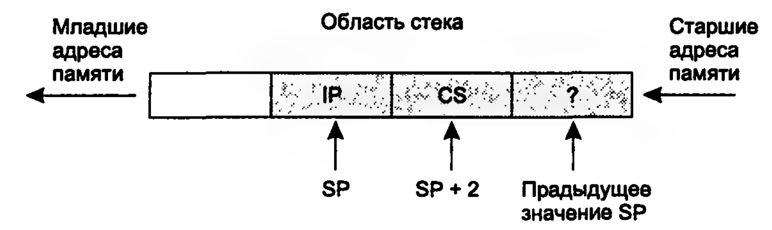



**Рис. 6.7.** Содержимое стека после вызова дальней процедуры

Поскольку процедура объявлена дальней (атрибут far), то команда ret имеет код 0CBh, отличный от кода аналогичной команды для вызова ближней процедуры (0C3h), и выполняется по-другому: из вершины стека извлекаются два слова и помещаются в регистры EIP и CS, передавая тем самым управление вызывающей программе из другого сегмента команд. Для команды возврата из дальней процедуры существует специальное мнемоническое обозначение retf.

Рассмотрим косвенный ближний вызов. В этом случае адрес процедуры содержится либо в ячейке памяти, либо в регистре. Это позволяет, как и в случае косвенного ближнего перехода, модифицировать адрес вызова, а также осуществлять вызов без использования метки по известному абсолютному адресу. Следующее 16-разрядное приложение иллюстрирует механизм косвенного вызова процедуры (листинг 6.4).

**Листинг 6.4.** Демонстрация косвенного вызова процедуры (16-разрядная версия)

```
.model small
.data
  s1     DB 0dh, 0ah, "Near indirect call of subr1 !$"
  s2     DB 0dh, 0ah, "Near indirect call of subr2 !$"
  addr1  DW subr1
  addr2  DW subr2
.code
 start:
  mov    AX, @data
  mov    DS, AX
  call   DS:addr1       ; вызов подпрограммы subr1 по смещению,
                        ; расположенному в переменной addr1
  call   DS:addr2       ; вызов подпрограммы subr2 по смещению,
                        ; расположенному в переменной addr2
  mov    AH, 1h
  int    21h
  mov    AX, 4C00h
  int    21h
 subr1   proc
   lea   DX, s1
   mov   AH, 9h
   int   21h
   ret
 subr1   endp
 subr2   proc
   lea   DX, s2
   mov   AH, 9h
   int   21h
   ret
 subr2   endp
 end     start
 end
```

Процедуры subr1 и subr2 с атрибутом near находятся в том же сегменте, что и вызывающая программа, а их относительные адреса — в переменных addr1 и addr2 в сегменте данных. Процедуры при вызове выводят соответствующие сообщения (строки s1 и s2) на экран.

Косвенный ближний вызов позволяет использовать разнообразные способы адресации процедур:

```
call BX        ; адрес подпрограммы находится в регистре BX
call[BX]       ; адрес подпрограммы находится в ячейке памяти, адрес
               ; которой помещается в регистр BX
call[BX][SI]   ; в BX адрес таблицы адресов подпрограмм,
               ; в SI индекс в этой таблице
tbl[SI]        ; переменная tbl содержит адрес таблицы адресов подпрограмм,
               ; в SI индекс в этой таблице
```

В листинге 6.5 приведен исходный текст 16-разрядного приложения, демонстрирующий один из вариантов реализации косвенного ближнего вызова. Здесь для вычисления смещения (эффективного адреса) процедуры используются

регистры SI и BX, причем в регистре SI содержится адрес таблицы tbl смещений подпрограмм, а регистр BX содержит индекс.

**Листинг 6.5.** Демонстрация косвенного ближнего вызова (16-разрядная версия)

```
.model small
data segment
 tbl label  word
     DW subrl          ; смещение процедуры subrl
     DW subr2          ; смещение процедуры subr2
     DW subr3          ; смещение процедуры subr3
 sl  DB 0dh, 0ah, "Near indirect call subrl demo 2 !$"
 s2  DB 0dh, 0ah, "Near indirect call subr2 demo 2 !$"
 s3  DB 0dh, 0ah, "Near indirect call subr3 demo 2 !$"
data ends
code segment
assume CS:code, DS:data
main proc
 mov   AX, data
 mov   DS, AX
 lea   SI, tbl           ; адрес таблицы смещений -> SI
 xor   BX, BX            ; начальное смещение -> BX
 mov   CX, 3             ; значение счетчика -> CX
next:
 call  word ptr [BX][SI] ; вызов одной из процедур
 add   BX, 2             ; индекс указывает на следующий элемент
                         ; в таблице смещений процедур
 dec   CX                ; уменьшить содержимое счетчика на 1
 jnz   next              ; следующая итерация
 mov   Ax, 4C00h
 int   21h
 main  endp
subrl  proc              ; объявление процедуры subrl
 lea   DX, sl
 mov   AH, 9h
 int   21h
 ret
subrl  endp
subr2  proc              ; объявление процедуры subr2
 lea   DX, s2
 mov   AH, 9h
 int   21h
 ret
subr2  endp
subr3  proc              ; объявление процедуры subr3
 lea   DX, s3
 mov   AH, 9h
 int   21h
 ret
subr3  endp
end    main
code   ends
end
```

Исходный текст программы несложен, хочу лишь обратить внимание на инструкцию

```
add   BX, 2
```

Эта инструкция находится в цикле

```
next:
 call  word ptr [BX][SI] ; вызов одной из процедур
 add   BX, 2
. . .
 jnz   next
```

Поскольку таблица tbl содержит смещения процедур в виде слов, то для перехода к следующему элементу таблицы содержимое BX увеличивается на 2.

Результатом выполнения подпрограммы будет вывод на экран строк:

```
Near indirect call subr1 demo 2 !
Near indirect call subr2 demo 2 !
Near indirect call subr3 demo 2 !
```

Последним мы рассмотрим косвенный дальний вызов. Его основное отличие от косвенного ближнего вызова состоит в том, что подпрограмма находится в другом сегменте, а в ячейке памяти содержится полный адрес подпрограммы, включая сегмент и смещение.

Пример косвенного дальнего вызова приведен в листинге 6.6. Это 16-разрядное приложение, в основу которого положен исходный текст предыдущего примера.

**Листинг 6.6.** Демонстрация косвенного дальнего вызова (16-разрядная версия)

```
.model large
data segment
 tbl label  dword
    DD subr1                   ; дальний адрес процедуры subr1
    DD subr2                   ; дальний адрес процедуры subr2
    DD subr3                   ; дальний адрес процедуры subr3
 s1  DB 0dh, 0ah, "FAR INDIRECT CALL subr1 DEMO !$"
 s2  DB 0dh, 0ah, "FAR INDIRECT CALL subr2 DEMO !$"
 s3  DB 0dh, 0ah, "FAR INDIRECT CALL subr3 DEMO !$"
data ends
code0 segment
assume CS:code0, DS:data
 main proc
  mov  AX, data
  mov  DS, AX
  lea  SI, tbl                 ; адрес таблицы адресов процедур -> SI
  push SI                      ; сохраним для дальнейшего использования
                               ; заполняем таблицу адресов для каждой
                               ; из процедур subr1, subr2 и subr3
  mov  word ptr [SI], offset subr1 ; смещение процедуры subr1 -> первое
                               ; слово двухсловной ячейки памяти
  mov  AX, code1               ; адрес сегмента, где находится
                               ; процедура subr1 -> AX
  mov  word ptr [SI+2], AX     ; содержимое AX -> второе слово
                               ; переход к следующему элементу таблицы
                               ; и сохранение дальнего адреса процедуры
                               ; subr2 во втором двойном слове
  add  SI, 4
```

продолжение ⇨

**Листинг 6.6** (продолжение)

```
  mov  word ptr [SI], offset subr2
  mov  AX, code2
  mov  word ptr [SI+2], AX
                                    ; переход к следующему элементу таблицы
                                    ; и сохранение дальнего адреса процедуры
                                    ; subr3 в третьем двойном слове
  add  SI, 4
  mov  word ptr [SI], offset subr3
  mov  AX, code3
  mov  word ptr [SI+2], AX
  pop  SI                           ; восстанавливаем начальный адрес
                                    ; таблицы tbl
  xor  BX, BX                       ; подготавливаем регистр BX, который
                                    ; будет использован для индексации
                                    ; таблицы
  mov  CX, 3                        ; значение счетчика -> CX
next:
  call dword ptr [BX][SI]           ; дальний косвенный вызов процедур
                                    ; subr1, subr2 и subr3
  add  BX, 4                        ; переход к адресу следующей процедуры
                                    ; в таблице tbl
  dec  CX                           ; уменьшить счетчик на 1
  jnz  next                         ; следующая итерация, если
                                    ; CX не равен 0
  mov  AX, 4C00h
  int  21h
 main  endp
code1  segment
assume CS:code1
subr1  proc far                     ; объявление процедуры subr1
  lea  DX, s1
  mov  AH, 9h
  int  21h
  ret
subr1  endp
code1  ends
code2  segment
assume CS:code2
subr2  proc far                     ; объявление процедуры subr2
  lea  DX, s2
  mov  AH, 9h
  int  21h
  ret
subr2  endp
code2  ends
code3  segment
assume CS:code3
subr3  proc far                     ; объявление процедуры subr3
  lea  DX, s3
  mov  AH, 9h
  int  21h
  ret
subr3  endp
code3  ends
end main
end
```

Программа довольно сложная, поэтому остановимся на ней подробно.

Анализ процедуры начнем со структуры таблицы tbl. Эта таблица содержит дальние адреса трех процедур (subr1, subr2 и subr3), находящихся в трех разных сегментах кода (code1, code2 и code3). Каждый элемент таблицы представляет собой двойное слово. Младшее слово двухсловного элемента содержит смещение (эффективный адрес) процедуры, старшее — адрес сегмента программного кода, в котором данная процедура находится. Таким образом, в таблице tbl зарезервировано 12 байт памяти для адресов трех процедур.

Программа заполняет 4-байтовые ячейки памяти необходимой информацией так, как это делается, например, для процедуры subr2:

```
mov  word ptr [SI], offset subr2
mov  AX, code2
mov  word ptr [SI+2], AX
```

После заполнения таблицы нужной информацией основная программа (находящаяся в программном сегменте code0) в цикле next, состоящем из трех итераций, выполняет дальние косвенные вызовы каждой из процедур:

```
next:
  call dword ptr [BX][SI]
  add  BX, 4
  dec  CX
  jnz  next
```

Результатом работы программы является вывод следующих трех строк на экран:

```
FAR INDIRECT CALL subr1 DEMO !
FAR INDIRECT CALL subr2 DEMO !
FAR INDIRECT CALL subr3 DEMO !
```

Прежде чем закончить тему адресации процедур, хочу сделать некоторые замечания. Если вы работаете с 32-разрядными приложениями (используется директива .model flat), то понятия «дальний вызов» не существует. Приложение выполняется в едином линейном адресном пространстве размером вплоть до 4 Гбайт, где данные и код перемешаны, а сегментные регистры установлены в одно и то же значение. Все вызовы и команды переходов считаются ближними (атрибут near ptr). Для таких вызовов можно применять те же режимы, что и для ближних вызовов в 16-разрядных моделях памяти (прямой ближний и косвенный ближний), но использовать при этом 32-разрядные переменные и регистры.

Для иллюстрации вышеизложенного приведу фрагмент 32-разрядной программы, вычисляющей сумму и разность двух целых чисел с использованием двух процедур. Исходный текст программного кода показан в листинге 6.7.

Программный код включает в себя вызывающую процедуру _far_demo32 и вызываемые процедуры sub1 и sub2. Процедура sub1 вычисляет сумму чисел i1 и i2, помещая результат в младшее двойное слово переменной res. Процедура sub2 вычисляет разность тех же чисел и помещает результат в старшее двойное слово переменной res. Процедура _far_demo32 вызывает процедуры по адресу, находящемуся в регистре ESI. Регистр ESI получает его из таблицы tbl, содержащей соответствующие адреса в двухсловных переменных.

**Листинг 6.7.** Демонстрация косвенного ближнего вызова (32-разрядная версия)

```
.686
.model flat
option casemap: none
.data
  tbl label  dword
      DD sub1
      DD sub2
  i1  DD -39
  i2  DD 41
  res DD 2 DUP(0)
.code
_far_demo32 proc
  lea   ESI, tbl
  mov  [ESI], offset sub1
  mov  [ESI+4], offset sub2
  call  dword ptr [ESI]
  call  dword ptr [ESI+4]
  lea   EAX, res
  ret
_far_demo32 endp
sub1 proc
 clc
 mov EAX, i1
 adc EAX, i2
 mov res, EAX
 ret
sub1 endp
sub2 proc
 clc
 mov EAX, i1
 sbb EAX, i2
 mov res+4, EAX
 ret
sub2 endp
end
```

Процедура _far_demo32 возвращает в программу адрес переменной res, содержащей два двойных слова с результатами сложения и вычитания. Как видно из листинга, 32-разрядный код намного упрощает механизм вызова подпрограмм, поскольку отпадает необходимость в сегментации программы и данных, а это значительно повышает производительность программ в целом.

## 6.3. Параметры процедур и возвращаемые значения

Наиболее распространенным способом для передачи аргументов в подпрограммы в языке ассемблера является размещение аргументов в регистрах. При этом вызывающая программа записывает фактические параметры в регистры, а процедура извлекает их оттуда и использует в своей работе. Подобный метод очень эффективен, поскольку вызванная подпрограмма может непосредственно использовать

переданные значения, при этом обращение к регистрам выполняется значительно быстрее, чем обращение к памяти.

Должен заметить, что имеются и определенные ограничения на использование этого метода. Дело в том, что процессор имеет не так много регистров, в то время как чуть ли не в каждой команде требуется тот или иной регистр. Существует высокая вероятность того, что вызывающей программе и процедуре одновременно потребуются для работы одни и те же регистры, что усложнит ситуацию. Можно попытаться разработать приложение таким образом, чтобы вызывающая программа и процедура использовали разные регистры, хотя сделать это довольно сложно — опять-таки по причине их ограниченного количества.

Простейший вариант избежать подобной ситуации — сохранить регистры в стеке при входе в подпрограмму и восстановить их при выходе из подпрограммы. Какие же регистры процессора можно использовать для передачи параметров и возврата результата? Программист может выбирать те или иные регистры, исходя из своих соображений, но и здесь есть некоторые правила.

Чаще всего для передачи параметров применяют регистры EAX, EBX, ECX, EDX, немного реже — EBP, ESI, EDI. Регистр EBP обычно используется вместе с регистром указателя стека ESP для доступа к параметрам, находящимся в стеке, и об этом мы поговорим отдельно. Регистры ESI и EDI удобны при выполнении операций с массивами данных в качестве индексных, хотя можно применять их по своему усмотрению.

Проиллюстрируем сказанное примером. Для простоты будем считать наш программный код 32-разрядным и использовать регистры и переменные той же разрядности. Предположим, что нужно найти меньшее из двух целых чисел и вычислить абсолютную величину (модуль) этого минимума. Для решения этой задачи разработаем две процедуры на ассемблере: minint — для вычисления минимума и minabs — для определения его абсолютного значения.

Процедура minint в качестве параметров принимает два целочисленных значения, процедура minabs в качестве единственного параметра — одно целочисленное значение.

Условимся первый параметр процедуры minint передавать в регистре EAX, а второй — в регистре EBX. Предположим, что у нас имеется два целых числа, расположенные в переменных i1 и i2. Кроме того, создадим переменные min_val и abs_val для сохранения минимума и модуля числа соответственно. Обе процедуры возвращают результат в регистре EAX. Результат выполнения процедуры minint является входным параметром для процедуры minabs.

После всех допущений и предположений исходный текст фрагмента программы может выглядеть так, как показано в листинге 6.8.

**Листинг 6.8.** Демонстрация передачи параметров процедуре через регистры (32-разрядная версия)

```
. . .
.data
  i1      DD 34
  i2      DD 17
  min_val DD ?
  abs_val DD ?
. . .
```

продолжение ⇗

**Листинг 6.8** (продолжение)

```
.code
. . .
 mov    EAX, i1
 mov    EBX, i2
 call   minint
                                  ; минимум двух чисел i1 и i2 находится в регистре EAX.
                                  ; Сохраним это значение в переменной min_val
                                  ; и найдем абсолютное значение
 mov    min_val, EAX
 call   minabs
                                  ; сохраним модуль числа в переменной abs_val
 mov    abs_val, EAX
  . . .
                                  ; здесь обьявляются процедуры minint и minabs
minint  proc
 cmp    EAX,EBX
 jl     exit
 mov    EAX,EBX
exit:
 ret
minint  endp
minabs  proc
 mov    EAX, min_val
 cmp    EAX, 0
 jge    quit
 neg    EAX
quit:
 ret
minabs  endp
. . .
```

При создании этого программного кода мы не делали никаких предположений относительно сохранности содержимого регистров. Если вызывающая программа в момент передачи управления процедуре minint использовала регистр EBX, то его содержимое может быть разрушено вызываемой процедурой. Чтобы избежать этого, нужно сохранить регистр EBX в стеке:

```
mov  EAX, i1
push EBX
mov  EBX, i2
call minint
pop  EBX
```

Сохранение регистров желательно делать в любой процедуре, даже если очевидно, что основная программа не использует те же регистры, что и процедура. Программный код в дальнейшем может измениться (что происходит очень часто), и может оказаться так, что после этих изменений основной программе потребуются эти регистры. Лучше всего предусмотреть сохранение всех регистров, воспользовавшись специальными командами pusha, pushad, popa и popad.

Замечу, что сохранять значение регистра, посредством которого процедура возвращает результат (обычно это AX или EAX), не нужно, поскольку в изменении этого регистра и заключается цель работы процедуры.

Передача параметров через регистры — удобный метод и используется очень часто. Он является эффективным, когда число параметров невелико; если же параметров много, то для них просто не хватит регистров. В таком случае реализуют другой способ передачи параметров — через стек: основная программа записывает фактические параметры (их значения или адреса) в стек, а процедура затем их оттуда извлекает. Это наиболее распространенный способ, применяемый в большинстве программ.

Как процедура получает доступ к параметрам? Общепринятым для этих целей считается регистр EBP. В него необходимо поместить адрес вершины стека (на него указывает регистр ESP), а затем использовать выражения вида [EBP+*n*] для доступа к параметрам процедуры. Максимальное значение числа *n* определяется количеством параметров и должно быть кратным 2 (2, 4, 6, 8 и т. д.). При этом желательно сохранить регистр EBP, поскольку он может потребоваться в основной программе.

Рассмотрим пример процедуры (sub2), в которой требуется найти разность двух целых чисел, причем параметры этой процедуре передаются через стек. Как обычно, процедура возвращает результат операции в регистрс EAX. Положим, что вызывающая программа передает процедуре в качестве параметров целочисленные переменные i1 и i2, а результатом выполнения процедуры будет разность i1 – i2.

Первое, что должна сделать вызывающая программа, — поместить передаваемые параметры в стек. Затем процедура должна извлечь их из стека и обработать. Вот исходный текст программного кода:

```
   . . .
.model flat
.data
  i1 DD 34
  i2 DD 190
.code
   . . .
   push i2
   push i1
   call sub2
   pop  i1
   pop  i2
   . . .
 sub2 proc
   push EBP
   mov  EBP, ESP
   mov  EAX, dword ptr [EBP+8]
   sub  EAX, dword ptr [EBP+12]
   pop  EBP
   ret
 sub2 endp
   . . .
```

Проанализируем исходный текст. В программном сегменте с помощью следующих команд параметры i1 и i2 помещаются в стек:

```
push i2
push i1
call sub2
```

Затем вызывается процедура sub2. Содержимое стека после выполнения этих команд будет таким, как показано на рис. 6.8.

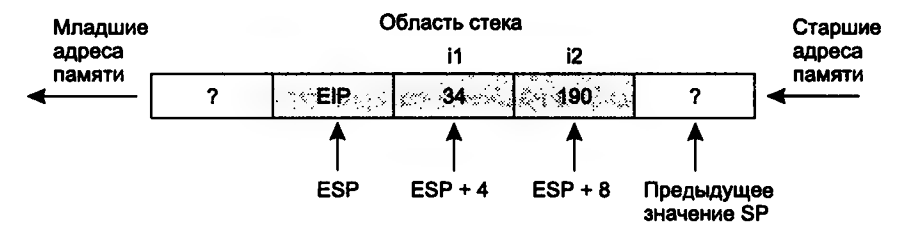



**Рис. 6.8.** Состояние стека после выполнения команд push i2 и push i1

Поскольку программа оперирует двойными словами, то содержимое указателя стека после выполнения команд push смещается на 8. Очередная команда call sub2 помещает в стек адрес команды, которая будет выполняться следующей. После входа в процедуру sub2 при помощи команды push EBP в стеке сохраняется содержимое регистра EBP, который должен использоваться для доступа к параметрам i1 и i2 в стеке. Область стека будет выглядеть так, как показано на рис. 6.9.

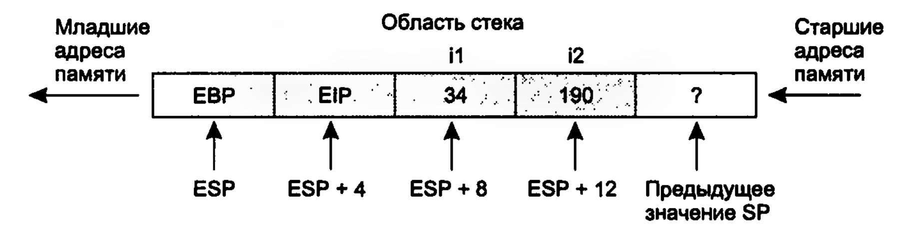



**Рис. 6.9.** Состояние стека после выполнения команд call sub2 и push EBP

Теперь, например, чтобы обратиться к переменной i1, следует в качестве одного из операндов указать [EBP+8], а переменная i2 будет находиться по адресу [EBP+12]. Как видим, первая помещенная в стек переменная имеет наибольший адрес, а последняя — наименьший. Следующие две инструкции ассемблера выполняют вычитание i2 из i1, то есть i1 – i2:

```
mov  EAX, dword ptr [EBP+8]
sub  EAX, dword ptr [EBP+12]
```

Результат вычитания остается в регистре EAX и возвращается вызывающей программе. Предпоследняя команда pop EBP восстанавливает содержимое регистра EBP, а команда ret передает управление инструкции, следующей за командой call, путем извлечения содержимого стека адреса этой инструкции и помещения этого адреса в регистр EIP.

Таким образом, после завершения процедуры sub2 в стеке остаются переменные i1 и i2. Вызывающая программа может хранить определенные данные в стеке по определенным смещениям относительно вершины стека, но проблема в том, что указатель стека не соответствует тому, который нужен программе. Если происходит обращение к стеку вызывающей программы, то немедленно наступает

крах. Поэтому очень важно восстановить или, по-другому, очистить стек до того состояния, какое он имел перед вызовом процедуры. Очистить стек может как вызывающая программа, так и процедура.

Простейший вариант — после вызова sub2 выполнить две команды pop в обратном порядке, что и сделано в этом фрагменте кода:

```
pop  i1
pop  i2
```

Эти команды никакой полезной функции, кроме восстановления указателя стека, не выполняют. Вызывающая программа может очистить стек и более изящным способом, сразу же после команды call используя команду

```
add ESP, 8
```

В результате выполнения этой команды указатель стека ESP сместится на два двойных слова, что нам и нужно. Тогда последовательность команд вызова процедуры и восстановления стека будет иметь вид

```
push i2
push i1
call sub2
add  ESP, 8
```

Процедура sub2 может и сама восстановить стек при выходе. Это можно сделать с помощью специальной команды ret *n*, где *n* — количество байтов, подлежащих удалению из стека. Программный код процедуры sub2 в этом случае изменится:

```
sub2 proc
 push EBP
 mov  EBP, ESP
 mov  EAX, dword ptr [EBP+8]
 sub  EAX, dword ptr [EBP+12]
 pop  EBP
 ret  8
sub2 endp
```

Команда ret — это одна из модификаций команды ret *n* при *n* = 0. При задании параметра *n* нужно помнить, что в операнде не должен учитываться адрес возврата — команда ret считывает его до очистки стека.

С точки зрения эффективности программы лучше, если очистку стека будет делать сама процедура. Если обращений к процедуре много, то в основной программе команду add придется использовать многократно, в то время как вызываемая процедура одна и в ней можно вызвать команду ret *n*. Это полезное правило для оптимизации программ: если какое-то действие может быть выполнено либо в основной программе, либо в процедуре, то лучше, если это будет сделано в процедуре. В этом случае требуется меньше команд.

Такова общая схема передачи параметров через стек. Еще раз напомню, что этот способ передачи параметров универсален, его можно использовать при любом числе параметров. Однако этот способ сложнее, чем передача параметров через регистры, поэтому желательно передавать параметры через регистры,

так проще и короче. Что же касается результата процедуры, то он крайне редко передается через стек и обычно передается через регистр. Общепринято, что результат выполнения процедуры возвращается в регистре EAX.

## 6.4. Использование общих переменных в процедурах

В большинстве случаев, особенно при разработке больших программ, возникает ситуация, когда требуется обрабатывать одни и те же объемы данных несколькими процедурами или даже отдельными программами. Очень удобно было бы сделать эти данные доступными для нескольких процедур. Может возникнуть вопрос: зачем применять для взаимодействия процедур какие-то специфические приемы, когда данные можно передавать от одной процедуры другой через параметры. Только что рассмотренные примеры как раз и демонстрировали такую методику. Однако в случае использования параметров как средства взаимодействия между процедурами возникают определенные проблемы. Перечислим только некоторые из них:

- При относительно большом количестве процедур обработки одних и тех же данных производительность процедуры снижается. Предположим, в процедуре имеется массив чисел, который обрабатывается несколькими процедурами. Каждый раз при обращении к процедуре другие программы вынуждены вычислять каким-то образом адреса элементов массива. Если используется стек, а в подавляющем большинстве случаев так оно и есть, возникает необходимость обращения к стеку для получения местоположения элементов массива в памяти. При относительно редком обращении к процедуре таких проблем может и не возникнуть, но с усложнением структуры программы и алгоритмов обработки быстродействие приложения может снизиться.
- При обработке одних и тех же данных разными процедурами структурируемость программы, использующей ассемблерные процедуры, ухудшается.

Общие или, как их еще называют, глобальные, переменные позволяют работать с данными при минимальном использовании стека, что экономит процессорное время. Кроме того, фиксированные привязки общих переменных на этапе компоновки ускоряют доступ к ним.

В дальнейшем мы будет употреблять термины «общая переменная» и «глобальная переменная» как синонимы. Для работы с общими переменными в языке ассемблера используются директивы public и extern. Директива public объявляет переменную или процедуру доступной для других модулей, директива extern указывает на то, что переменная или процедура является внешней по отношению к выполняемой процедуре. Обе директивы применяются для компоновки основной программы или процедуры из нескольких объектных модулей и очень удобны для построения больших программ.

Глобальные переменные объявляются следующим образом:

- в объектном модуле, где находится такая переменная, необходимо указать на возможность доступа к ней с помощью директивы public;
- в объектных модулях, из которых происходит обращение к общей переменной, необходимо объявить ее с директивой extern.

Использование общих переменных лучше показать на примере. Пусть требуется вычислить выражение $a_1 + a_2 - b_3$, где $a_1$, $a_2$ и $b_3$ — целые числа. Вычисление выполним с помощью двух процедур: _add2, которая вычисляет сумму $a_1 + a_2$, и _sub1, которая вычитает из этой суммы число $b_3$.

Результат вычислений возвращается в вызывающую процедуру (назовем ее _add_sub). Все три подпрограммы реализованы в виде отдельных файлов ASM, которые после компиляции образуют три файла объектных модулей с расширением OBJ. С помощью компоновщика (link или другого) эти файлы можно включить в 32-разрядное приложение.

В подпрограмме _add_sub определим переменные a1, a2 и b3 как двойные числа, которым присвоим конкретные значения. Поскольку все переменные должны быть доступны из других подпрограмм, находящихся в других объектных файлах, необходимо их объявить с директивой public. Кроме того, внешние подпрограммы _add2 и _sub2, используемые процедурой _add_sub, должны быть объявлены с директивой extern. Исходный текст подпрограммы _add_sub показан в листинге 6.9.

**Листинг 6.9.** Демонстрация использования общих переменных (32-разрядная версия)

```
.686
.model flat
.stack 100h
option casemap: none
  extern _add2:proc
  extern _sub1:proc
  public a1, a2, b3
.data
  a1    DD  12
  a2    DD  17
  b3    DD  34
.code
 _add_sub proc
   clc          ; очищаем флаг переноса
   call _add2   ; вычисляем сумму a1 + a2
   push  EAX    ; промежуточный результат помещаем в стек,
                ; поскольку он будет использоваться
                ; процедурой _sub1
   call _sub1   ; вычисляем разность полученной суммы (a1 + a2)
                ; и числа b3. Результат возвращается в основную
                ; программу через регистр EAX
   ret
 _add_sub endp
 end
```

Обратите внимание на директивы, выделенные жирным шрифтом. Если не указать процедуры как внешние (extern), то компилятор, не обнаружив их в текущем модуле, выдаст ошибку. Если не указать переменные с атрибутом public, то при компиляции этого ASM-файла никаких ошибок не будет, зато они появятся в процессе компиляции других модулей, где эти переменные используются.

Далее представлен исходный текст процедуры _add2:

```
.686
.model flat
option casemap: none
  public _add2
  extern a1: DWORD
  extern a2: DWORD
.code
 _add2 proc
   mov  EAX, a1
   adc  EAX, a2
   ret
 _add2 endp
 end
```

Поскольку процедура _add2 используется в подпрограмме _add_sub, то она должна быть объявлена как общая (public) для внешних по отношению к данному объектному файлу программ. Кроме того, процедура _add2 задействует две переменные, определенные в другом объектном файле (где определена процедура _add_sub), поэтому они должны быть указаны как внешние (extern). Все эти директивы выделены жирным шрифтом.

Исходный текст самой процедуры понятен, поэтому останавливаться на нем мы не будем. Замечу лишь, что результат сложения возвращается, как обычно, в регистре EAX.

Проанализируем последнюю из трех подпрограмм — _sub2. Исходный текст процедуры:

```
.686
.model flat
option casemap: none
  extern b3: DWORD
  public _sub1
.code
 _sub1 proc
   push  EBP
   mov   EBP, ESP
   mov   EAX, dword ptr [EBP+8]
   sub   EAX, b3
   pop   EBP
   ret  4
 _sub1 endp
 end
```

Процедура использует переменную b3 из модуля, в котором находится подпрограмма _add_sub, поэтому она объявлена как внешняя. Сама процедура _sub2

объявлена доступной для программ из других модулей, поскольку вызывается из внешней подпрограммы _add_sub. Эти директивы выделены жирным шрифтом.

Остановимся на программном коде процедуры _sub2 более подробно. Вспомним, что эта процедура вызывается из _add_sub следующим образом:

```
push  EAX
call _subl
```

Процедуре передается один параметр через стек (сумма a1 и a2). Для извлечения параметра из стека в процедуре _sub2 используется регистр EBP (мы рассматривали эту методику ранее). Перед возвратом в вызывающую подпрограмму в стеке находится один параметр, помещенный туда при вызове, поэтому команда ret должна очистить стек. Требуется удалить 4 байта, что и делает команда ret 4.

Как и в предыдущих случаях, процедура возвращает результат в регистре EAX. Таким образом, вызывающая подпрограмма _add_sub перед инструкцией возврата будет содержать результат в регистре EAX. При указанных значениях переменных регистр EAX будет содержать значение –5. Несмотря на простоту приведенного примера, он демонстрирует основные аспекты использования общих переменных в программах на ассемблере.

# Операции со строками и массивами



Большинство команд ассемблера оперируют байтом, словом или двойным словом. Однако во многих случаях бывает необходимо переслать или сравнить поля данных, которые превышают по размеру слово или байт. Например, может потребоваться сравнивать описания или имена, чтобы отсортировать их в определенной последовательности. Элементы такого формата известны как строковые данные и могут иметь как символьный, так и числовой тип.

Преимущества ассемблера проявляются и при обработке строк и массивов данных. Под операциями обработки строк мы будем понимать следующие операции:

- сравнение двух строк;
- копирование строки-источника в строку-приемник;
- считывание строк из устройства или файла;
- запись строки в устройство или файл;
- определение размера строки;
- нахождение подстроки в заданной строке;
- объединение двух строк (конкатенация).

Операции над строками широко используются в языках высокого уровня. Ассемблерная реализация таких операций позволяет существенно повысить быстродействие программ, особенно если требуется обработать большое количество строк и массивов.

Строка символов или чисел, с которыми программа работает как с группой, является обычным типом данных. Программа пересылает строку из одного места в другое, сравнивает ее с другими строками, ищет в ней заданное значение. При работе со строками программист сталкивается с необходимостью определить окончание строки, чтобы точно знать, когда заканчивать обработку. Существует два принципиально разных подхода к идентификации строки и ее элементов.

Можно указать размер строки (количество элементов, входящих в строку), записав число элементов в первый байт строки. По общепринятым соглашениям первый элемент строки имеет смещение 0, поэтому можно сказать, что размер строки прописывается в нулевом элементе, а символы строки начинаются с первого элемента. Такой принцип был реализован в языке Pascal и в среде программирования Delphi. Такие строки называются короткими (short strings), поскольку их размер не превышает 255 байт.

Наибольшее распространение получил второй способ идентификации строки, при котором в конце строки указывается нулевой символ (0). Такие строки называются строками с завершающим нулем (null-terminated strings). Они используются в языке C и в операционных системах Windows. Вот как выглядит такая строка на языке ассемблера:

```
String_0  DB "NULL-TERMINATED STRING".0
```

В языке ассемблера не существует каких-то стандартов для идентификации строк. Размер строки можно указать, используя нехитрый трюк. Лучше всего показать это на примере:

```
. . .
.data
 s1  DB "STRING"
 len EQU $-s1
. . .
```

Здесь определена строка символов s1, а ее размер len равен разности начального и конечного адресов элементов. Такой вариант очень удобен, поскольку константу len можно использовать для циклической обработки элементов строки. При этом len помещается в счетчик символов (обычно регистр CX или ECX, хотя могут быть и другие регистры).

Ничто не мешает использовать и строки с завершающим нулем, при этом в процессе обработки придется отслеживать конец строки. Вот фрагмент программного кода, демонстрирующий эту возможность:

```
. . .
.data
 s1  DB "TEST STRING".0
.code
. . .
lea  ESI. s1            : адрес первого элемента строки
. . .
cmp  byte ptr [ESI].0   : проверка на конец строки
. . .
```

Если использовать формат коротких строк (стиль Pascal), то обработку элементов можно организовать так, как показано в следующем примере:

```
. . .
.data
 s1 DB 7. "STRING1"
.code
. . .
```

```
lea ESI, s1
mov CL, byte ptr [ESI]
inc ESI
. . .
```

В этом случае обработка строки начинается с элемента с индексом 1, то есть находящегося по адресу [ESI+1]. В регистр CL помещается размер строки s1, равный 7 (команда mov CL, byte ptr [ESI]). Определенным недостатком такого метода является необходимость заранее знать размер строки.

До сих пор мы рассматривали строки, состоящие из символов, но наши рассуждения применимы и к последовательности произвольных байтов, слов и двойных слов. В этом случае такую совокупность элементов называют массивом. Для байтовых массивов действительны те же приемы работы, что и для символьных строк, а вот при работе с элементами размером в слово или двойное слово следует учитывать некоторые особенности, связанные с размерностью элементов.

Рассмотрим следующий пример. Пусть имеется массив целых чисел, представленных двойными словами. В этом случае для правильной обработки элементов в цикле счетчик в регистре CX (ECX) должен содержать количество двойных слов, а не байтов:

```
. . .
.data
 num_array DD 34, 456, -768, 12
 len       EQU $-num_array
.code
. . .
lea        ESI, dword ptr num_array ; адрес первого элемента -> ESI
mov        ECX, len                 ; размер массива в байтах -> ECX
shr        ECX, 2                   ; преобразовать в размерность
                                    ; двойных слов
. . .
add        ESI, 4                   ; переход к следующему элементу массива
. . .
```

Помимо коррекции счетчика необходимо правильно указывать адрес следующего элемента массива. Для двойного слова следующий элемент отстоит от предыдущего на 4 байта, для слова — на 2 байта.

Для работы со строками и массивами в систему команд процессоров Intel включены специальные команды обработки строк. Эту группу команд в терминологии Intel называют командами строковых примитивов, или цепочечными командами. Рассмотрим принципы работы цепочечных команд.

Цепочечная команда может быть использована для многократной обработки одного байта, одного слова или двойного слова. Для этого указывается префикс повторения rep. Далее приведены модификации префикса rep для команд строковых примитивов:

- rep — повторять операцию, пока CX не станет равным 0;
- repz, repe — повторять операцию, пока элементы равны, то есть до первого неравенства (флаг ZF установлен в 0). Операция прекращается, если флаг ZF устанавливается в 1 или счетчик в регистре ECX (CX) достигает нуля;

- repne, repnz — повторять операцию, пока элементы не равны, то есть до первого равенства (флаг ZF установлен в 1). Операция прекращается при установке флага ZF в 0 или при достижении значения 0 в регистре ECX (CX).

Для процессоров Intel, обрабатывающих слово за одну операцию, использование цепочечных команд там, где это возможно, повышает эффективность программы.

В строковых командах не применяются способы адресации, характерные для остальных команд обработки строк. Строковые команды адресуют операнды комбинациями регистров ESI(SI) или EDI (DI).

Операнды источника используют регистр ESI (SI), а операнды приемника (результата) — регистр EDI (DI). Все строковые команды корректируют адрес после выполнения операции. Строка может состоять из нескольких элементов, но команды обработки строк могут обрабатывать только один элемент в каждый момент времени. Автоматический инкремент (увеличение) или декремент (уменьшение) адреса операнда позволяет быстро обрабатывать строковые данные. Флаг направления DF в регистре состояния определяет направление обработки строк.

Если он равен 1, то адрес уменьшается, а если он сброшен в 0, то адрес увеличивается. Сама величина инкремента или декремента адреса определяется размером операнда. Например, для символьных строк, в которых размер операндов равен 1 байт, команды обработки строк изменяют адрес на 1 после каждой операции. Если обрабатывается массив целых чисел, в котором каждый операнд занимает 4 байта, то строковые команды изменяют адрес на 4. После выполнения операции указатель адреса в регистрах ESI (SI) или EDI (DI) ссылается на следующий элемент строки.

Мы будет рассматривать в основном строки с завершающим нулем. Можно выделить пять основных команд для работы со строками. К ним относятся:

- movs — команда перемещения строки данных из одного участка памяти в другой;
- lods — команда загрузки в регистр-аккумулятор EAX (AX, AL) строки, адрес которой указан в регистре ESI (SI);
- stos — команда сохранения содержимого регистра EAX (AX, AL) в памяти по адресу, указанному в регистре EDI (DI);
- cmps — команда сравнения строк, расположенных по адресам, содержащимся в регистрах ESI (SI) и EDI (DI);
- scas — команда сканирования строк, которая сравнивает содержимое регистра EAX (AX, AL) с содержимым памяти, определяемым регистром EDI (DI).

Каждая команда обработки строк имеет три допустимых формата. Например, команда movs может иметь одно из представлений: movsb, movsw или movsd. Команда movsb служит только для работы с однобайтовыми операндами, movsw — для работы со словами, а movsd — для работы с двойными словами. Суффиксы b, w и d определяют шаг инкремента и декремента для индексных регистров ESI (SI) и EDI (DI). Если команда используется в общем формате, то размерность операндов должна быть определена явно.

Перед выполнением команд строковых примитивов необходимо загрузить в регистры ESI (SI) и/или EDI (DI) адреса обрабатываемых ячеек памяти.

Обработку строк и массивов не обязательно выполнять с помощью команд строковых примитивов, но использовать такие команды во многих случаях удобнее. Рассмотрим более подробно операции со строками и начнем с пересылки и копирования данных.

Для иллюстрации операций обработки строк и массивов приводятся исходные тексты 16-разрядных и 32-разрядных процедур. Для лучшего понимания материала большинство 32-разрядных процедур не получают никаких параметров, а оперируют данными, определенными внутри самой процедуры. Результат работы процедур возвращается в регистре EAX либо в виде указателя (адреса) результата, либо как непосредственное значение.

# 7.1. Пересылка и копирование данных

Для выполнения операций пересылки и копирования строк и массивов очень удобно использовать команду movs. Эта команда может применяться для пересылки одиночных байтов, слов или двойных слов, однако с префиксом rep и счетчиком байтов в регистре ECX (CX) можно выполнять пересылку любого числа символов более эффективно. При этом в регистре EDI (DI) должен содержаться относительный адрес области памяти, в которую будет помещена строка, а адресация источника выполняется через регистр ESI (SI).

Таким образом, перед выполнением команды movs следует инициализировать регистры ESI (SI) и EDI (DI) требуемыми относительными адресами источника и приемника. В зависимости от состояния флага DF команда movs производит увеличение или уменьшение на 1 (для байта), на 2 (для слова) и на 4 (для двойного слова) содержимого регистров EDI (DI) и ESI (SI).

При разработке 16-разрядных приложений для адресации строки-источника используется пара регистров DS:SI, для адресации строки-приемника — ES:DI, а в качестве счетчика — регистр CX. В 32-разрядных приложениях для адресации используются регистры ESI и EDI, а в качестве счетчика — регистр ECX. Команды строковых примитивов можно заменить другими командами, причем иногда такая замена позволяет повысить производительность программного кода — этот и другие подобные вопросы мы рассмотрим в конце главы.

При использовании команды movsb, movsw или movsd компилятор ассемблера предполагает наличие корректного размера строковых данных и не требует кодирования операндов в команде. Для команды movs размер должен быть закодирован в операндах. Например, если переменные STRING_A и STRING_B определены как байтовые с помощью директивы DB, то следующая команда будет выполнять пересылку байтов, количество которых определено в регистре CX, из переменной STRING_B в переменную STRING_A:

```
rep movs STRING_A. STRING_B
```

Эту команду можно записать в альтернативной форме:

```
rep movs ES:BYTE PTR[DI]. DS:[SI]
```

В любом случае перед началом операции следует поместить в регистры DI и SI адреса STRING_A и STRING_B.

Префикс rep обеспечивает повторение команды несколько раз, и его нужно указывать непосредственно перед командой. Для использования префикса rep необходимо установить начальное значение в регистре CX, в этом случае при выполнении строковой команды (не только movs) происходит декремент регистра CX до нуля. Флаг направления DF определяет направление повторяющейся операции:

- для увеличивающихся адресов необходимо с помощью команды cld установить флаг DF в 0;
- для уменьшающихся адресов необходимо с помощью команды std установить флаг DF в 1.

Можно обойтись и без команд строковых примитивов при копировании данных, а использовать обычные команды ассемблера. Вот программный эквивалент команды rep movsb (для 16-разрядных операндов):

```
. . .
  mov  CX, counter
next:
  mov  AL,[SI]
  mov  DI],AL
  inc  SI | dec SI ; инкремент или декремент источника
  inc  DI | dec DI ; инкремент или декремент приемника
  loop next
. . .
```

Рассмотрим практический пример 16-разрядного приложения MS-DOS, выполняющего копирование строки с использованием команды movsb. Исходный текст программы приведен в листинге 7.1.

**Листинг 7.1.** Копирование символьных строк при помощи команды movsb (16-разрядная версия)

```
.model small
.data
  src   DB   "COPIED TEST STRING"
  len   EQU $-src
  dst   DB   len DUP (' ')
        DB   '$'
.code
start:
  mov  AX, @data   ; инициализация сегментных регистров
  mov  DS, AX
  mov  ES, AX
  cld              ; установим флаг направления DF для инкремента
  lea  SI, src     ; адрес источника -> DS:SI
  lea  DI, dst     ; адрес приемника -> ES:DI
  mov  CX, len     ; количество копируемых символов -> CX
  rep  movsb       ; копирование символов
  lea  DX, dst     ; отобразить скопированную строку на экране
  mov  AH, 9h
  int  21h
  mov  AX, 4c00h
  int  21h
  end  start
  end
```

Перед началом операции копирования необходимо установить флаг направления DF так, чтобы адреса источника и приемника увеличивались после каждой итерации. Для этого нужно установить флаг в 0 командой cld. Адрес строки-источника помещается в регистр SI, а адрес строки-приемника — в регистр DI. После копирования содержимое строки отображается на экране с помощью функции 9h прерывания 21h.

В 32-разрядных приложениях можно воспользоваться копирующей строки процедурой (назовем ее _cp_strings), показанной в листинге 7.2.

**Листинг 7.2.** Копирование символьных строк при помощи команды movsb (32-разрядная версия)

```
.586
.model flat
option casemap:none
.data
 s1  DB "TEST STRING TO COPY"
 len EQU $-s1
 s2  DB len DUP(' ')
.code
 _cp_strings proc
   cld
   lea  ESI, s1
   lea  EDI, s2
   mov  ECX, len
   rep  movsb
   lea  EAX, s2
   ret
  _cp_strings endp
  end
```

Операцию копирования строк здесь выполняет команда rep movsb, но для адресации строк используются 32-разрядные регистры ESI и EDI, поскольку в линейной модели адресации (директива .model flat) сегментные регистры не применяются. Процедура возвращает адрес строки-приемника в регистре EAX.

Операции копирования можно выполнять также и для массивов целых или вещественных чисел. Следующий фрагмент программного кода позволяет копировать содержимое целочисленного массива src_array в массив dst_array при помощи команды movsd:

```
. . .
.data
 src_array  DD 231, -12, 45, -65
 len        EQU $-src_array
 dst_array  DD 4 DUP (0)
.code
. . .
cld
mov  ECX, DWORD PTR lenarray
lea  ESI, DWORD PTR sarray
lea  EDI, DWORD PTR darray
rep  movsd
. . .
```

Команда movs может использоваться для еще одной весьма полезной операции над двумя строками, называемой сцеплением или конкатенацией. При ее выполнении к строке-приемнику добавляются символы строки-источника. При этом программист должен сам позаботиться о достаточном размере буфера приемника. Нужно учесть, что буфер приемника должен иметь размер, как минимум, равный сумме размеров сцепляемых строк. В листинге 7.3 приводится исходный текст 16-разрядной программы конкатенации строк.

**Листинг 7.3.** Сцепление символьных строк с использованием команды movsb (16-разрядная версия)

```
.model small
.data
 src       DB "ADDED CHARACHTERS$"     ; строка src, которая будет
                                       ; добавлена к dst
 len_src   EQU $-src                   ; размер строки src
 dst       DB "ORIGINAL CHARACHTERS"   ; строка-приемник, в конец которой
                                       ; будут добавлены
                                       ; символы строки src
 len_dst   EQU $-dst                   ; размер строки dst
 suppl     DB len_src+1 DUP(' ')       ; зарезервированная область памяти
                                       ; для размещения символов из строки
                                       ; src и символа пробела для
                                       ; разделения строк
.code
  start:
  mov   AX, @data
  mov   DS, AX
  mov   ES, AX
  cld
  lea   SI, src
  lea   DI, dst+len_dst+1
  mov   CX, len_src
  rep   movsb
  lea   DX, dst
  mov   AH, 9h
  int   21h
  mov   AX, 4c00h
  int   21h
  end   start
  end
```

В этой программе к содержимому строки dst добавляется содержимое строки src. Результирующая строка размещается по адресу dst и выводится на экран. Обратите внимание на количество зарезервированных байтов памяти для строки-приемника. Результат работы программы выводится на дисплей в виде строки

```
ORIGINAL CHARACHTERS ADDED CHARACHTERS
```

В листинге 7.4 представлен исходный текст 32-разрядной процедуры конкатенации строк (она называется _concat_strings).

Как и в предыдущих примерах, процедура возвращает адрес результирующей строки в регистре EAX.

**Листинг 7.4.** Сцепление символьных строк с использованием команды movsb (32-разрядная версия)

```
.586
.model flat
option casemap:none
.data
 src        DB "ADDED CHARACHTERS"
 len_src    EQU $-src
 dst        DB "ORIGINAL CHARACHTERS"
 len_dst    EQU $-dst
 suppl      DB len_src DUP(' ')
.code
 _concat_strings proc
   cld
   lea  ESI, src
   lea  EDI, dst+len_dst
   mov  ECX, len_src
   rep  movsb
   lea  EAX, dst
   ret
 _concat_strings endp
  end
```

Конкатенация массивов целых чисел или чисел с плавающей точкой выполняется по схожей схеме, необходимо лишь учитывать размер операндов. Принимающий массив должен иметь необходимое пространство для добавления новых элементов из массива-источника. Необходимое смещение в массиве-приемнике должно пересчитываться с учетом размерности в байтах элемента массива. Исходный текст процедуры (она называется _concat_dd), выполняющей конкатенацию двух массивов целых чисел, представлен в листинге 7.5.

**Листинг 7.5.** Объединение двух целочисленных массивов с использованием команды movsd (32-разрядная версия)

```
.586
.model flat
option casemap:none
.data
  i1          DD 23, 44, 8
  copy_area   DD 4 DUP (0)
  i2          DD -56, 7, -3, 89
  len         EQU $-i2
.code
 _concat_dd proc
   mov   ECX, len
   shr   ECX, 2
   lea   EDI, copy_area
   lea   ESI, i2
   cld
   rep   movsd
   lea   EAX, i1
   ret
 _concat_dd endp
 end
```

В этом фрагменте кода элементы массива-источника i2 записываются в массив-приемник i1 начиная с четвертого элемента. В регистр ESI помещен адрес массива i2, а регистр EDI содержит адрес дополнительной области памяти copy_area, куда будет производиться копирование.

Вместо команды lea EDI, copy_area можно использовать команду lea EDI, i1 + 12. Здесь число 12 означает смещение в байтах от начала массива i1. После выполнения процедуры массив i1 будет содержать элементы 23, 44, 8, –56, 7, –3 и 89.

Анализ операций копирования был бы неполным, если бы мы не упомянули о возможностях процессоров Intel Pentium по обработке массивов данных большой размерности. Максимальный размер операнда, над которым можно выполнять элементарную операцию, равен двойному слову. Для обработки операндов большей размерности, например два двойных слова (учетверенное слово), необходимо применять специальные приемы, обрабатывая их как отдельные двойные слова.

Процессоры Intel Pentium обеспечивают возможность параллельной обработки целочисленных данных, имеющих разрядность до 64 бит, используя так называемую MMX-технологию. Мы будем подробно рассматривать принципы, положенные в основу MMX, в последующих главах, однако сейчас нас будет интересовать одна из команд MMX-расширения, очень полезная при копировании и пересылке данных в обычных операциях, а именно команда movq. С помощью этой команды можно копировать и перемещать 8-байтовые данные. В качестве одного из операндов (источника или приемника) команда обязательно должна использовать один из 8 специальных 64-разрядных регистров, которые в ассемблере MASM обозначаются как MM0 – MM7. Вторым операндом команды может служить 64-разрядная ячейка памяти.

Премущества использования команды movq обусловлены следующим:

- за один цикл можно копировать или перемещать вдвое больше данных, чем это позволяют обычные команды, такие, как mov или movsd;
- MMX-расширение представляет собой отдельный аппаратно-программный модуль процессора, функционирующий относительно независимо от остальных вычислительных модулей; это позволяет MMX-командам выполняться параллельно с обычными командами, что ускоряет обработку данных;
- при передаче данных полностью используются преимущества 64-разрядной шины процессоров Intel Pentium.

Хочу сделать очень важное замечание: для успешной компиляции модулей, включающих MMX-команды, необходимо, чтобы компилятор ассемблера поддерживал эти команды. Старые компиляторы MASM версий 6.13*xxxx* и 6.14*xxxx* работать не будут, поэтому нужно использовать компилятор версии 7.10*xxxx*, входящий в состав Microsoft Driver Development Kit для Windows XP или Windows Server 2003.

Сейчас мы посмотрим, как практически реализовать эффективное копирование данных с помощью MMX-команд. Наш пример представляет собой процедуру (назовем ее _copy_movq), выполняющую копирование элементов массива целых

чисел из одной области памяти в другую блоками по два элемента за одну итерацию. В качестве параметров процедура принимает:

- адрес массива-источника [EBP+8];
- адрес массива-приемника [EBP+12];
- размер массива в двойных словах [EBP+16].

Процедура не возвращает результат, модифицируя массив-приемник (напомню, что каждый элемент массива имеет размерность двойного слова). Кроме того, массив-приемник имеет размер, как минимум, равный размеру массива-источника. Исходный текст процедуры представлен в листинге 7.6.

**Листинг 7.6.** Копирование целочисленного массива при помощи MMX-команды movq (32-разрядная версия)

```
.686
.model flat
option casemap: none
.MMX
.code
_copy_movq proc
  push EBP
  mov  EBP, ESP
  mov  ESI, dword ptr [EBP+8]     ; адрес массива-источника -> ESI
  mov  EDI, dword ptr [EBP+12]    ; адрес массива-приемника -> EDI
  mov  EAX, dword ptr [EBP+16]    ; размер массива-источника в двойных
                                  ; словах -> EAX
  xor  EDX, EDX                   ; регистр EDX участвует в операции
                                  ; деления, поэтому обнуляем его
  mov  EBX, 2
  div  EBX                        ; вычислим, сколько учетверенных слов
                                  ; помещается в массиве-источнике
  mov  ECX, EAX                   ; после деления: EAX = частное,
                                  ; EDX = остаток. Количество
                                  ; учетверенных слов -> ECX (счетчик
                                  ; цикла)
next:
  movq MM0, [ESI]                 ; копировать 8-байтовый операнд
                                  ; из массива-источника в регистр MM0
  movq [EDI], MM0                 ; копировать 8-байтовый операнд из
                                  ; регистра MM0 в массив-приемник
  add  ESI, 8                     ; адрес следующего 8-байтового элемента
                                  ; массива-источника -> ESI
  add  EDI, 8                     ; адрес следующего 8-байтового элемента
                                  ; массива-приемника -> EDI
  dec  ECX                        ; уменьшить счетчик цикла на 1
  jnz  next                       ; перейти к следующей итерации
  cmp  EDX, 0                     ; остались ли в массиве-источнике
                                  ; необработанные двойные слова?
  je   exit                       ; нет, выйти из процедуры
  mov  ECX, EDX                   ; да, повторить цикл для двойных слов
  cld                             ; флаг направления -> увеличение
                                  ; адресов
```

```
next_remainder:
  movsd                         ; скопировать двойное слово из
                                ; массива-источника в приемник
  dec  ECX                      ; декремент счетчика цикла
  jnz  next_remainder           ; если не равен 0, на следующую
                                ; итерацию
exit:
  pop  EBP
  ret
_copy_movq endp
   end
```

Остановимся на анализе некоторых важных моментов в выполнении процедуры. Команды movq в качестве одного из операндов используют MMX-регистр, в данном случае — регистр MM0. Хочу обратить ваше внимание на то, что принятое обозначение MMX-регистров (MM0 – MM7) в макроассемблере MASM совпадает с мнемоникой, принятой фирмой Intel. В других компиляторах ассемблера, вообще говоря, обозначения для MMX-регистров могут быть другими.

В процессе копирования команда movq оперирует двумя двойными словами (8 байт). Количество двойных слов в массиве-источнике может быть кратным или не кратным 2, и от этого будет зависеть способ копирования. При количестве элементов, кратном 2, все операции копирования могут быть выполнены командой movq, иначе потребуются дополнительные операции копирования, но с использованием обычных команд mov и movsd.

Например, если массив содержит 10 двойных слов, то количество учетверенных слов равно 5. В этом случае в счетчик основного цикла помещается число 5 и выполняется такое же количество операций копирования с помощью команд

```
movq MM0, [ESI]
movq [EDI], MM0
```

После этого происходит выход из процедуры. Представим ситуацию, когда массив содержит, например, 11 двойных слов. В этом случае количество учетверенных слов равно 5 плюс одно двойное слово. Следовательно, нужно обработать 5 учетверенных слов с помощью MMX-команд и одно двойное слово обычным образом. Для обработки оставшихся двойных слов используется дополнительный цикл из команд

```
next_remainder:
  movsd
  dec  ECX
  jnz  next_remainder
```

Определить, нужен ли дополнительный цикл, можно с помощью команд

```
cmp  EDX, 0
je   exit
```

И последнее. В исходном тексте процедур, использующих MMX-расширение, обязательно должна прсутствовать директива .MMX, позволяющая обрабатывать соответствующие команды. Хочу еще раз обратить внимание на то, что компиляторы MASM первых версий даже при наличии директивы .MMX команды этого расширения не обрабатывают!

Дальнейшим усовершенствованием процессоров Intel Pentium стало появление в составе процессора модуля SSE, позволяющего выполнять параллельную обработку чисел с плавающей точкой. Данная технология позволяет работать со 128-разрядными значениями, для чего в аппаратную архитектуру процессора включены восемь 128-разрядных регистров, которые обозначаются как XMM0 – XMM7. Мы рассмотрим более подробно эту технологию в следующих главах, сейчас же остановимся на одной из команд SSE-расширения, а именно — movups.

Эта команда чрезвычайно полезна для быстрого копирования и перемещения больших объемов данных. Она позволяет переслать 128-разрядный операнд из источника в приемник. В качестве источника и приемника может выступать либо один из регистров XMM, либо 128-разрядная ячейка памяти. При пересылке данных эта команда не требует выравнивания данных по 16-байтовой границе, что упрощает ее применение. Как и для только что рассмотренной команды movq, компилятор MASM должен поддерживать SSE-расширение (версия 7.10*xxxx*). Обозначения XMM-регистров (как и MMX) для компилятора MASM совпадают с мнемоникой Intel, но другие компиляторы могут иметь свои обозначения.

Команда movups является еще более мощной, чем movq, поскольку обеспечивает пересылку в два раза большего объема данных за одну операцию. Кроме того, поскольку модуль SSE является отдельным функционально законченным аппаратным узлом процессора, обработка данных в нем выполняется параллельно с центральным процессором, что значительно повышает производительность программ, работающих по этой технологии. Рассмотрим пример использования команды movups для быстрого копирования содержимого одной области памяти в другую. В исходном тексте ассемблерной процедуры (назовем ее _movups_copy) обязательно должна присутствовать директива .XMM. Программный код процедуры показан в листинге 7.7.

**Листинг 7.7.** Копирование данных 16-байтовыми блоками с помощью команды movups (32-разрядная версия)

```
.686
.model flat
.XMM
option casemap: none
.data
 a1  DD -7.8.-6.5.-3.-8.-3.33.21.-13.61.-1.11.-44.-970.-22.77.-901
 len EQU $-a1
 a2  DD len DUP(0)
.code
_movups_copy proc
  push EBX
  lea  ESI. a1
  lea  EDI. a2
  mov  ECX. len
  shr  ECX. 2
  mov  EBX. 4
  mov  EAX. ECX
  xor  EDX. EDX
  div  EBX
  mov  ECX. EAX
```

```
next_16:
  movups  XMM0, [ESI]
  movups  [EDI], XMM0
  add  ESI, 16
  add  EDI, 16
  dec  ECX
  jnz  next_16
  cmp  EDX, 0
  je   exit
  mov  ECX, EDX
  cld
  rep  movsd
exit:
  lea  EAX, a2
  pop  EBX
  ret
_movups_copy endp
end
```

Процедура позволяет выполнять копирование произвольного количества элементов из источника в приемник. В данной реализации источником данных является массив a1, содержащий 18 двойных слов, а приемником — массив a2. Для использования команды movups необходимо копировать по четыре двойных слова (16 байт) за одну операцию. Если массив двойных слов содержит число элементов, кратное четырем, например 4, 8, 32 и т. д., то операцию копирования можно выполнить в одном цикле, в котором счетчик цикла кратен четырем. Например, если в массиве имеется 8 двойных слов, то счетчик цикла нужно установить равным 2 и выполнить две операции копирования по 4 двойных слова.

Если массив содержит произвольное, не обязательно кратное четырем число элементов, как в нашем примере, то после окончания цикла копирования по 16 байт оставшиеся двойные слова можно скопировать командой movsd. Например, в нашем случае потребуется 4 цикла копирования (4 двойных слова × 4) плюс отдельное копирование двух двойных слов. Несмотря на необходимость выполнения дополнительных операций копирования, для больших массивов выигрыш в быстродействии будет очень большим, если используется команда movups! Например, для массива из 5987 двойных слов можно выполнить 1496 итераций с командой movups и отдельно скопировать 3 двойных слова командой rep movsd, поместив в счетчик на регистре ECX значение 3.

Для нашего случая выполняется 4 итерации копирования в цикле с меткой next_16 и копирование двух двойных слов с помощью команды rep movsd.

Последние две процедуры копирования с определенными изменениями могут быть использованы как в программах на ассемблере, так и в приложениях, написанных на языках высокого уровня (C++, Pascal). Здесь хотелось бы более подробно остановиться на этом вопросе.

О способах взаимодействия процедур на ассемблере, содержащихся в отдельных объектных модулях, и приложений на языках высокого уровня мы поговорим более подробно в следующих главах. Сейчас же нам нужны только самые необходимые сведения, чтобы понять смысл приведенного далее программного кода на языке высокого уровня (в данном случае Visual C++ .NET).

Проверку работоспособности 32-разрядного ассемблерного кода лучше и быстрее всего провести на работающем приложении, написанном на языке высокого уровня. Это позволяет легко получить визуальный результат работы программы, что чрезвычайно важно. Такое приложение может быть очень простым и состоять из нескольких строк программного кода, так, чтобы любой программист, в том числе начинающий, мог легко в нем разобраться и воспроизвести.

Лучшим выбором здесь является компилятор С++, вернее, версия Microsoft Visual C++ .NET. Хочу добавить, что при использовании других современных компиляторов языка С++ исходный текст приведенных программ не изменится. Здесь и в последующих главах для 32-разрядных процедур на ассемблере будут приводиться исходные тексты вызывающих их программ на С++.

Процедура на ассемблере и программа на языке высокого уровня взаимодействуют следующим образом:

- ассемблерная процедура должна быть предварительно объявлена в программе на языке высокого уровня, при этом должны указываться параметры и возвращаемое значение;
- параметры в вызываемую процедуру передаются через стек справа налево, то есть самый правый параметр имеет наибольшее смещение в стеке;
- процедура возвращает результат в регистре EAX (значение или адрес).

Параметры в процедуру можно передавать через стек, но для их извлечения удобнее использовать не регистр стека ESP, а регистр EBP. Процедуру _movup_copy несложно включить в простую консольную программу на Visual C++ .NET, исходный текст которой показан в листинге 7.8.

**Листинг 7.8.** Демонстрационная программа для процедуры _movups_copy из листинга 7.7

```
#include <stdio.h>
extern "C" int* movups_copy(void):
int main(void)
{
 int* pint - movups_copy():
 printf("movups example: "):
 for (int i1 - 0: i1 < 18: i1++)
  {
   printf("%d ". *pint++):
  }
 return 0:
}
```

Для использования процедуры в приложении на С++ нужно объявить ее внешней с директивой extern:

```
extern "C" int* movups_copy(void):
```

После этого требуется включить объектный модуль, содержащий процедуру _movups_copy, в состав проекта. Результатом выполнения программы будет вывод на консоль строки, отображающей элементы массива a2:

```
movups demo: -7 8 -6 5 -3 -8 -3 33 21 -13 61 -1 11 -44 -970 -22 77 -901
```

# 7.2. Сравнение строк и массивов

Операция сравнения строк и массивов может выполняться как с помощью обычных команд ассемблера, так и с использованием специальной цепочечной команды cmps. Команда cmps сравнивает содержимое одной области памяти (адресуемой регистрами DS:SI или ESI) с содержимым другой области памяти (адресуемой регистрами ES:DI или EDI). В зависимости от значения флага DF команда cmps также увеличивает или уменьшает адреса в регистрах ESI (SI) и EDI (DI) на 1 для байта, на 2 для слова и на 4 для двойного слова. Команда cmps воздействует на флаги AF, CF, OF, PF, SF и ZF. При использовании префикса rep регистр ECX (CX) должен содержать количество сравниваемых переменных, при этом команда cmps может сравнивать любое число байтов или слов.

При выполнении команды cmps (а также scas) возможна установка флагов состояния так, чтобы операция могла завершиться сразу после обнаружения необходимого условия.

Работу команды cmps демонстрирует следующий пример, представляющий собой 16-разрядное приложение MS-DOS. Программа сравнивает содержимое двух символьных строк и отображает результат сравнения на экране дисплея. Исходный текст программы показан в листинге 7.9.

**Листинг 7.9.** Сравнение двух символьных строк (16-разрядная версия)

```
.model small
.stack
.data
  src     DB    "FIRST sTRING 1"
  lsrc    EQU   $-src
  dst     DB    "FIRST STRING 1"
  equal   DB    "Strings are equal", '$'
  non_eq  DB    "Strings are not equal", '$'
.code
start:
   mov  AX, @data    ; инициализация сегментных регистров
   mov  DS, AX
   mov  ES, AX
   cld               ; установка флага напрвления DF для
                     ; инкремента адресов
   lea  SI, src      ; адрес источника -> DS:SI
   lea  DI, dst      ; адрес приемника -> ES:DI
   mov  CX, lsrc     ; количество сравниваемых байтов -> CX
   repe cmpsb        ; попарное сравнение байтов
   je   yes          ; строки одинаковы?
   lea  DX, non_eq   ; нет, отобразить соответствующее сообщение
   jmp  output
yes:                 ; да, отобразить сообщение
   lea  DX, equal
output:
   mov  AH, 9h
   int  21h
   mov  AX, 4c00h
   int  21h
   end  start
   end
```

Сравнение строк в 32-разрядном приложении выполнить сложнее. Соответствующий программный код реализован в виде процедуры _compare_strings, исходный текст которой представлен в листннге 7.10.

**Листинг 7.10.** Сравнение символьных строк (32-разрядная версия)

```
.586
.model flat
option casemap:none
.data
 not_equal DB "NOT equal"
           DB 0
 equal     DB "Equal"
           DB 0
.code
 _compare_strings proc
   push EBP
   mov  EBP, ESP
   mov  EDI, dword ptr [EBP+16] ; адрес первой строки -> EDI
   mov  ESI, dword ptr [EBP+12] ; адрес второй строки -> ESI
   mov  ECX, dword ptr [EBP+8]  ; размер строки -> ECX
   cld                          ; установить флаг направления
                                ; для инкремента адресов
   repe cmpsb                   ; сравнение элементов строк
   je   s_equals
   lea  EAX, not_equal
   jmp  exit
 s_equals:
   lea  EAX, equal
 exit:
   pop  EBP
   ret
  _compare_strings endp
  end
```

Проверить работоспособность процедуры можно на примере простой программы, написанной на Visual C++ .NET (листинг 7.11).

**Листинг 7.11.** Демонстрационная программа для процедуры копирования строк из листинга 7.10

```
#include <stdio.h>
#include <string.h>
extern "C" char* compare_strings(int len, char* src, char* dst);
int main(void)
{
char *src = "My String C";
char *dst = "My String c";
printf("Comparing result: %s\n", compare_strings(strlen(src), src, dst));
return 0;
}
```

При использовании команд cmps следует учитывать, что сами строки могут иметь разный размер. Если строка-источник меньше, чем строка-приемник, то флаг CF устанавливается в 1. Если строки равны, то флаг ZF = 1. Если строка-

источник больше, чем строка-приемник, то CF = 0 и ZF = 0. Правильно установить равенство-неравенство строк — задача непростая, и нужно быть очень внимательным при выборе критерия равенства. Далее рассмотрим исходный текст небольшой программы, позволяющей показать различные ситуации, возникающие в процессе сравнения. Это простое 16-разрядное приложение (листинг 7.12).

**Листинг 7.12.** Сравнение строк по различным критериям (16-разрядная версия)

```
.model small
.data
  src    DB "STRING 32X"
  dst    DB "STRING 32X"
  len    EQU $-dst
  eql    DB 0dh, 0ah, "Strings are equal$"
  neq    DB 0dh, 0ah, "String are different$"
  less   DB 0dh, 0ah, "Src less than Dst$"
  great  DB 0dh, 0ah, "Src great than Dst$"
  .code
  start:
  mov  AX, @data
  mov  DS, AX
  mov  ES, AX
  cld
  lea  SI, src
  lea  DI, dst
  mov  CX, len
  repe cmpsb
  je   s_eq
  jne  not_eq
  jmp  ex
s_eq:
  lea   DX, eql
  jmp  show
src_less:
  lea   DX, less
  jmp   show
src_gr:
  lea   DX, great
  jmp   show
not_eq:
  lea   DX, neq
  mov   AH, 9h
  int   21h
  jb    src_less
  ja    src_gr
  jmp   ex
show:
  mov   AH, 9h
  int   21h
ex:
  mov   AX, 4c00h
  int   21h
  end   start
  end
```

При том содержимом, что указано для строк src и dst, программа выдает сообщение

```
Strings are equal
```

Если мы изменим строку src, добавив в ее конец какой-нибудь символ, например X, то строки src и dst будут иметь разный размер, но по-прежнему при запуске программы будет выдаваться сообщение о равенстве строк. Дело в том, что количество сравниваемых символов len, которое помещается в регистр CX, остается неизменным и равным размеру строки dst. В этом случае первые len символов обеих строк одинаковы, поэтому результат сравнения остается неизменным.

Если при равенстве размеров строк хотя бы два символа будут различаться, то программа выдаст одно из следующих сообщений:

```
Strings are different !
Src is greater than Dst
```

или

```
Strings are different !
Src is less than Dst
```

Может возникнуть и ситуация, при которой строка-приемник будет больше по размеру, чем строка-источник, например:

```
.data
src  DB "STRING 32X"
dst  DB "STRING 32XW"
len  EQU $-dst
```

В этом случае при продвижении указателей адресов памяти, находящихся в регистрах DS:SI и ES:DI, последним элементом строки dst будет символ W, а последним элементом src — какое-либо произвольное значение из диапазона 0–255 (если интерпретировать байт как беззнаковое число). Результат сравнения, таким образом, окажется непредсказуемым (если до того не возникнет ошибка доступа к памяти).

Наилучшим выходом из такой ситуации, позволяющим избежать ошибок, может быть следующий алгоритм: вначале определяется размер строк источника и приемника, и, если они равны, выполняется операция сравнения. Если строки различаются по размеру, то понятно, что они не равны и сравнение проводить не нужно. В листинге 7.13 показан вариант 16-разрядного приложения, иллюстрирующего эту методику.

**Листинг 7.13.** Улучшенная версия программы из листинга 7.12

```
.model small
.data
  src  DB  "STRING 1"
  lens EQU $-src
  dst  DB  "STRING 1"
  lend EQU $-dst
  eql  DB 0dh, 0ah, "Strings are equal ! $"
  neq  DB 0dh, 0ah, "Strings are different !$"
  diff_len DB "Strings have a different length!$"
```

```
.code
 start:
  mov  AX, @data
  mov  DS, AX
  mov  ES, AX
  cld
  mov  AX, lens
  cmp  AX, lend
  je   go_compare
  lea  DX, diff_len
  jmp  show
go_compare:
  lea  SI, src
  lea  DI, dst
  mov  CX, lens
  repe cmpsb
  je   s_eq
  lea  DX, neq
show:
  mov   AH, 9h
  int   21h
  jmp   ex
s_eq:
  lea   DX, eql
  jmp  show
ex:
  mov   AX, 4c00h
  int   21h
  end   start
  end
```

В этой программе вначале производится сравнение размеров операндов:

```
mov  AX, lens
cmp  AX, lend
```

После этого, в зависимости от результата, выполняются соответствующие действия.

До сих пор мы рассматривали строки, размер которых определен заранее. Проанализируем, каким образом можно работать со строками другого типа, а именно со строками с завершающим нулем. Напомню, что в конце таких строк помещается 0, по которому и определяется конец строки.

Для того чтобы сравнивать такие строки в ассемблерных программах, нужно вначале определить размер строки, после чего перейти к операции сравнения. Ассемблерные процедуры сравнения строк с завершающим нулем часто используются программами на языках высокого уровня для быстрой обработки строк. Рассмотрим 16-разрядное приложение, демонстрирующее обработку строк с завершающим нулем. Исходный код программы представлен в листинге 7.14.

Исходный текст программы несложен и не нуждается в подробных объяснениях. Замечу лишь, что размер строк вычисляется как разность начального и конечного адресов: для строки-источника src начальный и конечный адреса определяются регистрами DX:CX, а для строки-приемника dst — регистрами BX:AX.

**Листинг 7.14.** Обработка строк с завершающим нулем (16-разрядная версия)

```
.model small
.data
 src        DB  "STRING 11", 0
 dst        DB  "STRING 117", 0
 s_eq       DB "Strings are equal$"
 s_ne       DB "Strings are not equal$"
 size_diff  DB "Strings have different size !$"
.code
start:
  mov  AX, @data
  mov  DS, AX
  mov  ES, AX
  lea  SI, src        ; начальный адрес строки-источника -> SI
  mov  DX, SI         ; сохранить начальный адрес в регистре DX
  mov  AL, 0          ; символ для сравнения -> AL
src_again:
  cmp  AL, [SI]       ; достигнут конец строки?
  je   check_dst      ; да, проверить строку-приемник
  inc  SI             ; нет, конец строки не обнаружен, перейти
                      ; к следующему адресу
  jmp  src_again
check_dst:
  lea  DI, dst        ; начальный адрес строки-приемника -> DI
  mov  BX, DI         ; сохранить начальный адрес в регистре BX
dst_again:
  cmp  AL, [DI]       ; достигнут конец строки?
  je   check_size     ; да, сравнить размеры строк
  inc  DI             ; нет, конец строки не обнаружен, перейти
                      ; к следующему адресу
  jmp  dst_again
check_size:
  mov  CX, SI         ; конечный адрес строки-источника -> CX
  sub  CX, DX         ; вычислить размер строки-источника как разность
                      ; конечного и начального адресов
  mov  AX, DI         ; конечный адрес строки-приемника -> AX
  sub  AX, BX         ; вычислить размер строки-источника как разность
  sub  AX, BX         ; конечного и начального адресов
  cmp  AX, CX         ; размеры строк равны?
  je   compare        ; да, сравним строки
  lea  DX, size_diff  ; нет, отобразить сообщение
  jmp  show
compare:
  cld                 ; установить флаг направления для инкремента адресов
  mov  SI, DX         ; начальный адрес строки-источника -> SI
  mov  DI, BX         ; начальный адрес строки-приемника -> DI
  repe cmpsb          ; сравнить строки
  je   equal          ; строки равны?
  lea  DX, s_ne       ; нет, отобразить соответствующее сообщение
  jmp  show
equal:
  lea  DX, s_eq       ; да, отобразить соответствующее сообщение
show:
```

```
mov  AH, 9h
int  21h
mov  AX, 4c00h
int  21h
end start
end
```

Подобный пример для 32-разрядных приложений немного сложнее. Рассмотрим исходный текст процедуры (назовем ее _cmpsb_32), принимающей в качестве параметров адреса строк и возвращающей адрес строки с результатом сравнения (листинг 7.15). Такую процедуру можно использовать как в 32-разрядном приложении на ассемблере, так и в программе на одном из языков высокого уровня (C++, Pascal).

**Листинг 7.15.** Сравнение строк (32-разрядная версия)

```
.686
.model flat
option casemap: none
.data
  s_eq      DB "Strings are equal",0
  s_ne      DB "Strings are not equal",0
  size_diff DB "Strings have different size !",0
.code
_cmpsb32 proc
  push  EBP
  mov   EBP, ESP
  mov   ESI, dword ptr [EBP+12] ; адрес строки-источника
  mov   EDX, ESI
  mov   EAX, 0
src_again:
  cmp   EAX, [ESI]
  je    check_dst
  inc   ESI
  jmp   src_again
check_dst:
  mov   EDI, dword ptr [EBP+8]  ; адрес строки-приемника
  mov   EBX, EDI
dst_again:
  cmp   EAX, [EDI]
  je    check_size
  inc   EDI
  jmp   dst_again
check_size:
  mov   ECX, ESI
  sub   ECX, EDX
  mov   EAX, EDI
  sub   EAX, EBX
  cmp   EAX, ECX
  je    compare
  lea   EAX, size_diff
  jmp   exit
compare:
  cld
```

продолжение ⇗

**Листинг 7.15** (продолжение)

```
  mov  ESI, EDX
  mov  EDI, EBX
  repe cmpsb
  je   equal
  lea  EAX, s_ne
  jmp  exit
equal:
  lea  EAX, s_eq
exit:
  pop  EBP
  ret
 _cmpsb32 endp
  end
```

Программа, вызывающая эту процедуру, может быть, например, такой (Visual C++ .NET), как показано в листинге 7.16.

**Листинг 7.16.** Демонстрационная программа для процедуры из листинга 7.15

```
#include <stdio.h>
extern "C" char* cmpsb32(char* src, char* dst);
int main(void)
{
 char *s1 = "String 31 ";
 char *s2 = "String 31";
 printf(": %s\n", cmpsb32(s1, s2));
 return 0;
}
```

Различные манипуляции со строками и массивами, как уже упомииалось, можно реализовать и без команд строковых примитивов. Не является исключением и сравнение строк. В альтериативном программном коде обычно используют команду cmp, с помощью которой в цикле выполняется попарное сравнение элементов строк. В качестве примера можно привести 16-разрядное приложение, исходный код которого представлен в листинге 7.17.

**Листинг 7.17.** Сравнение строк при помощи обычных команд ассемблера (16-разрядная версия)

```
.model small
.stack
.data
  src    DB    "FIRST STRING 1"
  lsrc   EQU $-src
  dst    DB    "FIRST STRING 2"
  equal  DB    "EQUAL", '$'
  non_eq DB   "NOT equal", '$'
.code
start:
   mov  AX, @data
   mov  DS, AX
   mov  ES, AX
   lea  SI, src
   lea  DI, dst
```

```
   mov  CX, lsrc
again:
   mov  AL, [SI]
   cmp  AL, [DI]
   je   next_cmp
   lea  DX, non_eq
   jmp  output
next_cmp:
   inc  SI
   inc  DI
   loop again
   lea  DX, equal
output:
   mov  AH, 9h
   int  21h
   mov  AX, 4c00h
   int  21h
   end  start
   end
```

Здесь сравнивается содержимое регистра AL, в который помещается байт операнда-источника, с содержимым операнда-приемника, адресуемого регистром EDI. В зависимости от результата сравнения происходит переход на нужную ветвь программы, что и выполняется следующим фрагментом кода:

```
mov AL, [SI]
cmp AL, [DI]
je  next_cmp
```

Операции сравнения выполняются в цикле, счетчик которого находится в регистре CX. Если произошел обычный выход из цикла (по достижении счетчиком значения 0), то это означает, что строки равны и можно вывести соответствующее сообщение:

```
. . .
loop again
lea  DX, equal
. . .
```

В последних поколениях процессоров появился ряд команд, позволяющих повысить производительность приложений. Мы рассматривали такие команды в главе 5. Следующий пример демонстрирует, как можно сравнить строки и одновременно обеспечить высокую скорость выполнения программ, если использовать одну из таких команд, а именно cmove. В этой процедуре (назовем ее _eq_bytes) присутствует только один условный переход, остальная часть программного кода выполняется линейно. Программный код процедуры сравнивает элементы строк s1 и s2 и сохраняет одинаковые символы строк в строке dst. В этой процедуре исключены лишние ветвления, что может обеспечить более высокую производительность работы программного кода.

Хочу уточнить, что команда cmove включена в систему команд процессоров Pentium II и выше, поэтому с более ранними моделями процессоров программный

код работать не будет. Исходный текст 32-разрядной процедуры показан в листинге 7.18.

**Листинг 7.18.** Выбор одинаковых элементов из двух символьных строк (32-разрядная версия)

```
.686
.model flat
option casemap: none
.data
  s1  DB "Test STRING"   ; первая строка
  len EQU $-s1           ; размер строки
  s2  DB "TEST STRING"   ; вторая строка
  dst DB 11 DUP(' '),0   ; результирующая строка
.code
 _eq_bytes proc
   push   EBX
   lea    ESI, s1              ; адрес первой строки -> ESI
   lea    EDI, s2              ; адрес второй строки -> EDI
   lea    EDX, dst             ; адрес результирующей строки -> EDX
   mov    ECX, len             ; размер строки -> ECX
 next:
   mov    BL, '-'              ; исходное значение -> BL (вместо
                               ; символа '-' можно взять другой)
   mov    AL, [ESI]            ; символ из строки-источника -> AL
   cmp    AL, [EDI]            ; сравнить с символом в той же позиции
                               ; строки-приемника
   cmove  EBX, EAX             ; если символы равны, поместить в регистр BL
                               ; символ из AL
   mov    byte ptr [EDX], BL   ; сохранить символ на соответствующей
                               ; позиции в результирующей строке
   inc    ESI                  ; адрес следующего символа источника -> ESI
   inc    EDI                  ; адрес следующего символа приемника -> EDI
   inc    EDX                  ; адрес следующего символа
                               ; строки результата -> EDX
   loop   next                 ; переход к следующей итерации
   lea    EAX, dst             ; адрес результирующей строки -> EAX
   pop    EBX
   ret
  _eq_bytes endp
  end
```

Алгоритм процедуры достаточно прост, хочу лишь заметить, что команда cmove не работает с 8-разрядными регистрами, поэтому пересылка данных происходит из 32-разрядного регистра.

Процедура возвращает в вызывающую программу адрес строки, содержащей результат. Результирующая строка dst после выполнения процедуры будет содержать строку

```
T--- STRING
```

Используя исходный код предыдущего примера, можно создать эффективную процедуру для сравнения элементов двух массивов целых чисел. Такую процедуру назовем _eq_dwords, а ее исходный текст представлен в листинге 7.19.

**Листинг 7.19.** Выбор одинаковых элементов из двух целочисленных массивов (32-разрядная версия)

```
.686
.model flat
option casemap: none
.data
  num1  DD 4562, 1094, -502, 902
  len   EQU $-num1
  num2  DD 2341, 1094, -502, 87
  dst   DD 4 DUP(0)
.code
 _eq_dwords proc
   push   EBX
   lea    ESI, num1
   lea    EDI, num2
   lea    EDX, dst
   mov    ECX, len
 next:
   mov    EBX, 0
   mov    EAX, [ESI]
   cmp    EAX, [EDI]
   cmove  EBX, EAX
   mov    [EDX], EBX
   add    ESI, 4
   add    EDI, 4
   add    EDX, 4
   loop   next
   lea    EAX, dst
   pop    EBX
   ret
 _eq_dwords endp
 end
```

Программный код процедуры достаточно понятен, хочу лишь обратить внимание на то, что переход к следующим элементам массивов требует смещения на 4 байта. Кроме того, при несовпадении элементов сравниваемых массивов в соответствующие позиции результирующего массива заносятся нули. После выполнения процедуры массив dst будет содержать следующие элементы:

```
0 1094 -502 0
```

# 7.3. Сканирование строк и массивов

В процессе обработки текстовой информации может возникнуть необходимость манипулирования элементами строк или массивов. Наиболее часто выполняются такие операции:

- замена определенных символов в тексте другими, например подстановка пробелов вместо различных редактирующих символов (табуляции, возврата каретки и перевода строки);

- подсчет количества элементов строк или массивов, удовлетворяющих какому-либо условию;
- обработка отдельного элемента массива чисел, например вычисление модуля числа, квадратного корня и т. д.

Язык ассемблера обладает широкими возможностями для выполнения таких операций. Мы рассмотрим различные способы сканирования строк и массивов и начнем с использования специально предназначенной для этого команды scas. Команда scas отличается от команды cmps тем, что выполняет просмотр строки или массива в поиске определенного элемента. Команда scas сравнивает содержимое области памяти, адресуемой регистрами ES:DI или EDI, с содержимым регистра AL, AX или EAX.

Операция сравнения осуществляется путем вычитания содержимого ячейки памяти из содержимого AL, AX или EAX, с устанавлением при этом соответствующих флагов. В зависимости от значения флага DF команда scas увеличивает или уменьшает адрес в регистре DI (EDI) на 1 для байта, на 2 для слова и на 4 для двойного слова. Команда scas устанавливает флаги AF, CF, OF, PF, SF и ZF. При использовании префикса rep и размера строки в регистре CX (ECX) команда scas может сканировать строки и массивы любого размера. Эта команда особенно полезна при разработке текстовых редакторов, где программа должна сканировать строки, выполняя поиск знаков пунктуации.

В приведенной в листинге 7.20 программе выполняется сканирование строки src, и символ пробела заменяется символом +. Программа реализована как 16-разрядное приложение и выводит на экран содержимое строки src после преобразования.

**Листинг 7.20.** Поиск символа пробела в строке (16-разрядная версия)

```
.model small
.stack 100h
.data
 src DB " TEST STRING !$"
 len EQU $-src
.code
 start:
   mov   AX. @data
   mov   ES. AX
   mov   DS. AX
   lea   DI. src
   mov   CX. len-1
   cld
   mov   AL. ' '
next:
   scasb
   je    change
   loop  next
   lea   DX. src
   mov   AH. 9h
   int   21h
   jmp   exit
```

```
change:
  mov     byte ptr [DI-1]. '+'
  dec     CX
  jmp     next
exit:
  mov     AX. 4c00h
  int     21h
  end     start
  end
```

Команды сканирования данных позволяют реализовывать довольно сложные алгоритмы простыми средствами. Предположим, нужно подсчитать количество слов в данном фрагменте текста при условии, что в качестве разделителя используется символ пробела. В листинге 7.21 приводится простое 16-разрядное приложение, позволяющее выполнить такой подсчет и вывести результат на экран дисплея.

**Листинг 7.21.** Подсчет количества слов во фрагменте текста (16-разрядная версия)

```
.model small
option casemap:none
.data
  s1  DB  " TEST: first word     second word    third     word OK     !  "
  len EQU $-s1
  msg DB  "Number of words = "
  cnt DB  0
      DB  '$'
.code
start:
  mov   AX. @data
  mov   DS. AX
  mov   ES. AX
  mov   CX. len      : размер строки -> CX
  lea   DI. s1       ; адрес первого элемента строки -> DI
  mov   AL. ' '      : символ-разделитель -> AX
  cld                : инкремент адреса для последующих операций
next:
  repe  scasb        : пропускаем пробелы
  je    exit         : кроме пробелов. ничего нет - закончить работу
                     : программы
  inc   cnt          : обнаружено слово - увеличить счетчик слов
  repne scasb        : ищем конец слова. дальше должны быть пробелы
  jne   exit         : строка закончилась - выйти из программы
  jmp   next         : поиск следующего слова
exit:
 or     cnt. 30h     ; преобразовать однобайтовое число
                     : в символьный формат ASCII
 lea    DX. msg      : отобразить результат
 mov    AH. 9h
 int    21h
 mov    AX. 4C00h    : завершить программу
 int    21h
end     start
end
```

При данных параметрах строки s1 программа отобразит сообщение

```
Number of words = 9
```

Следующий пример представляет собой 32-разрядную процедуру, выполняющую поиск в массиве двойных слов чисел, меньших 100, и замену таких чисел нулями. Исходный текст процедуры (она называется _scas_dd) представлен в листинге 7.22.

**Листинг 7.22.** Поиск и замена чисел, меньших 100, в массиве целых чисел (32-разрядная версия)

```
.586
.model flat
option casemap:none
.code
_scas_dd proc
 push   EBP
 mov    EBP, ESP
 mov    ECX, dword ptr [EBP+12] ; размер массива -> ECX
 mov    EDI, dword ptr [EBP+8]  ; адрес первого элемента -> EDI
 mov    EAX, 100                ; сравниваемое значение
 cld                            ; установить флаг направления в сторону
                                ; увеличения адресов
next:
 scasd                          ; сравниваем элементы массива
                                ; с содержимым EAX
 jg    change                   ; число в EAX больше текущего элемента?
                                ; если нет, следующая итерация
 loop  next
 jmp   exit
change:                         ; элемент массива меньше числа в EAX
 mov   dword ptr [EDI-4], 0     ; заменить содержимое ячейки памяти на 0
 dec   ECX                      ; декремент счетчика
 jmp   next
exit:
 pop   EBP
 ret
_scas_dd endp
end
```

В этой процедуре используется тот факт, что команда scas после выполнения устанавливает флаги CF, AF, ZF и SF, и это позволяет анализировать значения по принципу «больше или меньше».

Процедуру при желании можно легко модифицировать для работы со своими данными и задействовать как в программах на ассемблере, так и в приложениях, разработанных на языках высокого уровня. В листинге 7.23 показан пример 32-разрядного консольного приложения на Visual C++ .NET, использующего эту процедуру.

**Листинг 7.23.** Демонстрационная программа для процедуры из листинга 7.22

```
#include <stdio.h>
extern "C" void scas_dd(int* pi, int isize);
int main(void)
```

```
{
 int iarray[7] = {794, 56, 132, 112, 32, 7, 265};
 int* pint = iarray;
 scas_dd(iarray, sizeof(iarray)/4);
 for (int i1 = 0; i1 < sizeof(iarray)/4; i1++)
  {
   printf("%d\t", *pint++);
  }
 printf("\n");
 return 0;
}
```

Результатом работы этого приложения будет вывод на экран ряда чисел:

```
794 0 132 112 0 0 265
```

Можно повысить производительность этой процедуры, если уменьшить количество условных переходов. В листинге 7.24 показан модифицированный вариант процедуры _scas_dd.

**Листинг 7.24.** Улучшенный вариант процедуры из листинга 7.22

```
.686
.model flat
option casemap:none
.data
  iarray DD 23, -49, 65, 98, 133, 82 ; исходные значения элементов
                                     ; массива
  len    DD $-iarray                 ; размер массива в байтах
.code
_scas_dd proc
 mov    ECX, len                     ; размер массива -> ECX
 shr    ECX, 2                       ; преобразовать в количество
                                     ; двойных слов
 lea    EDI, iarray                  ; адрес массива -> EDI
 mov    EAX, 100                     ; шаблон для сравнения -> EAX
 xor    EDX, EDX                     ; подготовка регистра EDX
 cld                                 ; установить флаг направления
                                     ; для увеличения адреса
next:
 mov    EBX, dword ptr [EDI]         ; элемент массива -> EBX
 scasd                               ; сравнить EAX с элементом массива
 cmovl  EBX, EDX                     ; если элемент массива больше 100,
 mov    dword ptr [EDI-4], EBX       ; обнулить его. Поскольку указатель
                                     ; адреса после выполнения команды
                                     ; scasd продвинулся на 4, необходимо
                                     ; это учесть в команде mov
 loop  next
exit:
 lea   EAX, iarray                   ; адрес массива -> EAX
 ret
_scas_dd endp
end
```

По сравнению с предыдущей процедурой здесь сделаны следующие изменения:

- другое условие задачи — теперь все элементы массива, большие 100, заменяются нулями;
- вместо команд условных переходов используется команда cmovl;
- вместо внешних параметров процедура использует данные из сегмента данных (массив iarray).

При указанных начальных значениях элементов iarray после выполнения процедуры в массиве будут содержаться следующие элементы:

```
23 -49 65 98 0 82
```

## 7.4. Использование команд lods и stos

Команда lods может загрузить из памяти один байт в регистр AL, одно слово в регистр AX или одно двойное слово в регистр EAX. Адрес памяти определяется либо регистрами DS:SI, либо регистром ESI. Как и для предыдущих цепочечных команд, в зависимости от значения флага DF происходит инкремент или декремент содержимого регистра SI или ESI. Эта команда используется без префикса повторения rep.

Если сравнивать команды mov и lods, то при одинаковом результате mov требует трех байтов машинного кода, в то время как lods — только одного. Команда lods дополнительно требует инициализации регистра SI или ESI. Команда lods удобна в тех случаях, когда нужно анализировать каждый байт, слово или двойное слово и выполнять с ними определенные действия. Команда lods подойдет, например, при замене отдельных символов строки, для реверсирования строки, то есть изменения порядка следования элементов на обратный, и т. д. Команда lods, как и остальные цепочечные команды, имеет модификации для работы с байтами (lodsb), со словами (lodsw) и двойными словами (lodsd).

Команду lods и ее модификации можно представить в виде нескольких других команд. Например, команду lodsb можно представить как комбинацию команд:

```
mov  AL,[SI]
inc  SI
```

Команда stos записывает содержимое одного из регистров AL, AX или EAX в байт, слово или двойное слово в памяти. Адрес памяти определяется регистрами ES:DI либо регистром EDI. В зависимости от значения флага DF команда stos увеличивает или уменьшает адрес в регистре DI на 1 для байта, на 2 для слова и на 4 для двойного слова. Команда stos, используемая с префиксом rep, позволяет инициализировать область данных какими-либо значениями, например пробелами или нулями. Команда stos, как и остальные цепочечные команды, имеет модификации для работы с байтами (stosb), со словами (stosw) и двойными словами (stosd).

Команду stos и ее модификации можно представить в виде нескольких других команд. Например, команда stosb может быть представлена в виде двух команд:

```
mov [DI], AL
inc DI
```

Следующие команды эквивалентны команде rep stosb:

```
. . .
next:
 mov    [DI], AL        ; содержимое AL -> ячейка памяти с адресом в DI
 inc    DI  |  dec  DI  ; инкремент или декремент
 loop   next            ; перейти к следующему элементу
. . .
```

Очень часто команды lods и stos используют вместе для посимвольной обработки строк и массивов. Наш следующий пример демонстрирует это (листинг 7.25). Это простая программа, в которой символы строки s1 из нижнего регистра преобразуются к верхнему регистру.

**Листинг 7.25.** Преобразование символов нижнего регистра в символы верхнего регистра (16-разрядная версия)

```
.model small
option casemap:none
.data
  s1    DB " test string 1 "
  len   EQU $-s1
        DB  '$'
.code
start:
  mov   AX, @data
  mov   DS, AX
  mov   ES, AX
  cld             ; установить флаг направления в сторону увеличения
                  ; адресов
  mov   CX, len   ; размер строки s1 -> CX
  lea   SI, s1    ; адрес первого элемента строки -> SI
  mov   DI, SI    ; тот же адрес -> DI
next:
  lodsb           ; загрузить символ строки s1 в регистр AL
  cmp   AL, 97    ; AL < 'a'?
  jb    skip      ; нет, вне диапазона, пропустить
  cmp   AL, 122   ; AL > 'z'?
  ja    skip      ; нет, вне диапазона, пропустить
  sub   AL, 32    ; преобразовать символ из диапазона 'a' - 'z'
                  ; в символ из диапазона 'A' - 'Z'
skip:
  stosb           ; запомнить символ по тому же адресу
  loop  next      ; переход к следующему символу
  jmp   exit
exit:
 lea    DX, s1    ; отобразить преобразованную строку
 mov    AH, 9h
 int    21h
 mov    AX, 4C00h
 int    21h
end     start
```

Команды lods и stos можно использовать и для копирования строк, однако такой метод не очень эффективен. Более удобной в этом плане является команда пересылки movs. Она считывает данные по адресу памяти, находящемуся в регистре SI или ESI, и помещает их по адресу, указываемому регистром DI или EDI. При этом содержимое регистров SI (ESI) и DI (EDI) изменяется так, чтобы указывать на следующие элементы строк. Кроме того, команда movs не загружает регистр-аккумулятор во время пересылки. Только movs и еще одна строковая команда cmps работают с двумя операндами памяти. Все остальные команды требуют, чтобы один или оба операнда находились в одном из регистров процессора.

Рассмотрим еще один пример применения команды lods. Процедура _count_b, исходный текст которой представлен в листинге 7.26, выполняет подсчет числа вхождений определенного символа в строке. Для этого примера подсчитывается количество символов s.

**Листинг 7.26.** Подсчет количества определенных символов в строке (32-разрядная версия)

```
.686
.model flat
option casemap: none
.data
 src   DB "This string contains five words"
 len   EQU $-src
 cnt   DD 0
.code
 _count_b proc
   lea   ESI, src
   mov   ECX, len
   cld
   xor   EBX, EBX
 next:
   lodsb
   cmp   AL, 's'
   sete  BL
   add   cnt, EBX
   loop  next
   lea   EAX, cnt
   ret
 _count_b endp
 end
```

В этой процедуре подсчет символов выполняется для строки src. Использование команды sete позволило избежать дополнительных ветвлений в программе после выполнения операции сравнения. Для подсчета количества вхождений символа задействован счетчик cnt. Он инкрементируется всякий раз, как только в процессе просмотра встречается символ s. После выполнения процедуры в счетчике будет находиться значение 3.

В следующем примере продемонстрирован один из способов изменения порядка следования элементов в строке на обратный (реверсирование строки). Во время этой операции последний элемент строки помещается на место первого, предпоследний — на место второго и т. д. Существуют различные способы выпол-

нения такой операции. В нашем случае выберем простой вариант, при котором для хранения промежуточного результата используется дополнительный буфер памяти. В листинге 7.27 представлен исходный текст 16-разрядного приложения, в котором применяется подобная техника преобразования.

**Листинг 7.27.** Реверсирование строки (16-разрядная версия)

```
.model small
.data
 src  DB "123 456 789$"  ; строка, которую нужно реверсировать
  len EQU $-src-1        ; размер строки за вычетом '$'
  tmp DB 11 DUP (20h)    ; временный буфер для хранения данных
.code
 start:
   mov  AX, @data
   mov  DS, AX
   mov  ES, AX
   mov  CX, len
   std                    ; флаг DF -> 1, уменьшение адреса src
   lea  SI, src           ; адрес преобразуемой строки -> SI
   add  SI, len-1         ; установить указатель на адрес последнего
                          ; элемента строки src
   lea  DI, tmp           ; адрес временного буфера памяти -> DI
next:
   lodsb                  ; элемент строки src -> AL
   mov  byte ptr [DI], AL ; сохранить символ в буфере tmp
   inc  DI                ; перейти к следующему адресу
                          ; в буфере tmp (инкремент адреса)
   loop next              ; следующая итерация
   cld                    ; установить флаг DF в 0 для
                          ; увеличения адресов
   mov  CX, len           ; размер строки src -> CX
   lea  SI, tmp           ; адрес строки-источника -> SI
   lea  DI, src           ; адрес строки-приемника -> DI
   rep  movsb             ; копирование из tmp в src
   lea  DX, src           ; вывод результата на экран
   mov  AH, 9h
   int  21h
   mov  AX, 4c00h
   int  21h
   end  start
   end
```

Исходный текст программы несложен для понимания, поэтому остановлюсь лишь на некоторых моментах. Для выполнения строковых команд в сторону уменьшения адресов операндов флаг направления DF должен быть установлен в 1. Кроме того, нужно правильно установить адрес последнего операнда (первого при выполнении операции).

Рассмотрим еще один пример работы команды stos (листинг 7.28). Эта команда очень удобна для инициализации области памяти каким-либо значением. В примере показано заполнение символьной строки символом X.

**Листинг 7.28.** Заполнение области памяти определенным символом (16-разрядная версия)

```
.model small
.data
  s1  DB " TEST STRING$"
  len EQU $-s1
.code
  start:
   mov   AX, @data
   mov   DS, AX
   mov   ES, AX
   cld            ; установить флаг переноса для инкремента адреса
   mov  AL, 'X'   ; символ-заполнитель -> AL
   lea  DI, s1    ; адрес строки-приемника
   mov  CX, len-1 ; размер строки без учета последнего символа -> CX
   rep  stosb     ; заполнить область памяти символом 'X'
   lea  DX, s1    ; вывод обновленной строки на экран
   mov  AH, 9h
   int  21h
   mov  AX, 4c00h
   int  21h
   end  start
   end
```

Исходный текст программы прост и в пояснениях не нуждается. Массив целых чисел, представленных элементами размером в слово, можно проинициализировать нулями с помощью следующего фрагмента программного кода:

```
 . . .
.data
  iarray   DW 34, -16, 8, 33, 92, 14
  len      EQU $-iarray
.code
  . . .
 lea  DI, iarray
 mov  CX, len
 shr  CX, 1
 cld
 mov  AX, 0
 rep  stosw
 . . .
```

# 7.5. Массивы строк

Во многих случаях программисту приходится иметь дело не с отдельными строками, а с группой, или массивом, строк. Очень часто в программах требуется, в зависимости от результатов каких-то промежуточных вычислений, отображать соответствующую информацию, содержащуюся в одной из строк массива. Доступ к отдельным строкам массива лучше всего продемонстрировать на примере (листинг 7.29). Это простое 16-разрядное приложение, выводящее на экран строку, адрес которой определяется переменной num.

**Листинг 7.29.** Обработка отдельных строк из группы строк (16-разрядная версия)

```
.model small
.stack 100h
.data
  s1      DB 0dh, 0ah, "String s1$"  ; содержимое строки s1
  s2      DB 0dh, 0ah, "String s2$"  ; содержимое строки s2
  s3      DB 0dh, 0ah, "String s3$"  ; содержимое строки s3
  saddr  label dword                 ; адрес массива строк saddr
          DW  s1                     ; адрес строки s1
          DW  s2                     ; адрес строки s2
          DW  s3                     ; адрес строки s3
  num     DW  0                      ; начальное значение счетчика
.code
 start:
  mov   AX, @data
  mov   DS, AX
  mov   ES, AX
  mov   CX, 3
again:
  xor   SI, SI                       ; подготовить индексный регистр SI
  add   SI,  num                     ; num -> SI
  shl   SI, 1                        ; вычислить смещение
                                     ; в массиве saddr
  mov   DX, word ptr saddr[SI]       ; поместить в DX адрес строки
                                     ; и вывести ее на экран
  mov   AH, 9h
  int   21h
  inc   num                          ; инкремент счетчика для перехода
                                     ; к следующей строке
  loop  again                        ; следующая итерация
exit:
  mov   AX, 4c00h
  int   21h
  end start
  end
```

Программа очень простая, единственный момент, на котором я хочу акцентировать внимание, — алгоритм вычисления адреса строки. Каждая строка имеет двухбайтовый адрес (для данной программы!), поэтому массив строк saddr содержит три слова. Например, для получения адреса строки s2 необходимо к адресу массива saddr прибавить 2, а для получения адреса s3 — 4, адрес строки s1 имеет нулевое смещение. Для вывода строки на экран нужно в регистр EDX загрузить адрес массива saddr плюс смещение строки в массиве. Смещение находится в регистре ESI и легко вычисляется с помощью команд

```
xor   SI, SI
add   SI,  num
shl   SI, 1
```

Например, при значении num = 0 регистр SI будет содержать значение 0, что даст доступ к строке s1, при значении num = 1 в SI будет находиться 2 и т. д.

Принципы работы с массивами строк для 32-разрядных приложений на ассемблере и языках высокого уровня демонстрирует процедура _sarray, исходный код которой показан в листинге 7.30.

**Листинг 7.30.** Обработка группы строк (32-разрядная версия)

```
.686
.model flat
.data
  s1  DB "It's a String s1",0
  s2  DB "Here is String s2",0
  s3  DB "String s3 is placed here",0
  saddr   label dword
      DD  s1
      DD  s2
      DD  s3
 .code
 _sarray proc
   push EBP
   mov  EBP, ESP
   mov  ECX, dword ptr [EBP+8] ; индекс строки -> ECX
   shl  ECX, 2                 ; преобразовать в двойное слово
   lea  ESI, saddr             ; адрес массива строк saddr -> ESI
   add  ESI, ECX               ; адрес двойного слова, содержащего
                               ; адрес строки -> ESI
   mov  EAX, [ESI]             ; адрес искомой строки -> EAX
   pop  EBP
   ret
 _sarray endp
 end
```

Процедура принимает в качестве параметра номер строки, равный 0, 1 или 2, и возвращает адрес выбранной ранее строки. В качестве индексатора строки выбран регистр ECX, в котором полученный номер строки умножается на 4 (адрес строки представлен двойным словом). В регистр ESI помещается адрес массива строк (он совпадает с адресом двойного слова, содержащего адрес нулевой строки). Для доступа к двойному слову, содержащему адрес требуемой строки, следует к содержимому ESI прибавить содержимое регистра ECX. Наконец, в регистр EAX помещается адрес искомой строки.

Эта процедура может быть вызвана из программы на Visual C++ .NET с помощью программного кода из листинга 7.31.

**Листинг 7.31.** Демонстрационная программа для процедуры из листинга 7.30

```
#include <stdio.h>
extern "C" char* sarray(int i1);
int main(void)
{
 for (int i1 = 0; i1 < 3; i1++)
  {
   printf(": %s\n", sarray(i1));
  }
 return 0;
}
```

В этой программе в цикле for все три строки выводятся на экран дисплея.

# 7.6. Полезные алгоритмы

Здесь мы рассмотрим некоторые алгоритмы, которые могут пригодиться при работе со строками и массивами, а также ознакомимся с некоторыми командами ассемблера, полезными при обработке строк.

Ранее мы рассматривали алгоритм реверсирования строк. Суть алгоритма состоит в том, что вначале меняются местами первый и последний элементы массива, затем — второй и предпоследний и т. д. Проход по массиву (строке) выполняется с помощью двух указателей — в прямом и обратном направлениях. Обмен прекращается, когда значение прямого указателя становится равным или большим обратного указателя.

Приведу пример программы, реализующей этот алгоритм. В программе используется новая для нас команда — xchg. Команда xchg пересылает значение первого операнда во второй, а второго — в первый. В качестве первого операнда можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго — регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Эта команда очень удобна при перемещении значений и обмене значениями между операндами.

В листинге 7.32 приводится исходный код 16-разрядного приложения, в котором выполняется реверсирование элементов строки.

**Листинг 7.32.** Реверсирование строки при помощи команды xchg (16-разрядная версия)

```
.model small
.data
  s1   DB "ABCDEFG$"
  len  EQU $-s1-1          ; в константе len не учитывается последний
                           ; элемент ('$'), поскольку он остается на
                           ; месте
.code
 start:
  mov    AX, @data
  mov    DS, AX
  mov    ES, AX
  lea    SI, s1            ; адрес первого
                           ; элемента -> SI (прямой указатель)
  lea    DI, s1+len-1      ; адрес символа 'G' -> DI (обратный
                           ; указатель)
next:
  mov    AL, byte ptr [SI] ; здесь выполняется обмен элементов строки,
  xchg   AL, byte ptr [DI] ; находящихся в ячейках памяти с адресами
  mov    byte ptr [SI], AL ; в регистрах SI и DI
  inc    SI                ; продвинуть вперед прямой указатель
  dec    DI                ; уменьшить обратный указатель
  cmp    SI, DI            ; сравнить адреса
  jb     next              ; адрес в SI все еще меньше адреса в DI
                           ; повторить цикл
  lea    DX, s1            ; преобразование закончено, вывод результата
  mov    AH, 9h
```

продолжение ⇨

**Листинг 7.32** (продолжение)

```
  int    21h
  mov    AX, 4C00h
  int    21h
  end    start
  end
```

Подобным образом можно выполнить, например, реверсирование массива целых чисел. В этом случае необходимо учитывать, что следующие адреса элементов отстоят на 4 от предыдущих. В листинге 7.33 показан пример простой процедуры (она называется _rev32), меняющей порядок следования элементов массива целых чисел на обратный.

**Листинг 7.33.** Реверсирование массива целых чисел при помощи команды xchg (32-разрядная версия)

```
.686
.model flat
option casemap: none
.code
 _rev32 proc
   push  EBP
   mov   EBP, ESP
   mov   ECX, dword ptr [EBP+12] ; помещаем размер массива
                                 ; в двойных словах в регистр ECX
   dec   ECX                     ; вычисляем смещение последнего элемента
   shl   ECX, 2                  ; преобразуем смещение в байты
   mov   ESI, dword ptr [EBP+8]  ; адрес массива
   mov   EDI, ESI
   add   EDI, ECX
next:
   mov   EAX, [ESI]
   xchg  EAX, [EDI]
   mov   [ESI], EAX
   add   ESI, 4
   sub   EDI, 4
   cmp   ESI, EDI
   jb    next
   pop   EBP
   ret
 _rev32  endp
  end
```

Полагаем, что процедура принимает два параметра: адрес массива (он будет находиться в регистре EBP со смещением 8) и размер массива (в регистре EBP со смещением 12). Здесь размер массива передается в двойных словах, поэтому для вычисления правильного смещения последнего элемента массива необходимо выполнить команды

```
dec   ECX
shl   ECX, 2
```

Процедура _rev32 не возвращает результат, а работает с буфером вызывающей программы. На языке Visual C++ .NET результат преобразования выводится с помощью консольного приложения, исходный текст которого представлен в листинге 7.34.

**Листинг 7.34.** Демонстрационная программа для процедуры из листинга 7.33

```
include <stdio.h>
extern "C" void rev32(int* iarray, int isize);
int main(void)
{
 int iarray[6] = {45, -234, 12, 98, 33, 2};
 int isize = sizeof(iarray) / 4;
 int* pia = iarray;
 rev32(iarray, 6);
 for (int i1 = 0; i1 < isize; i1++)
  {
   printf("%d\t", *pia++);
  }
  return 0;
}
```

Довольно часто программисты сталкиваются с необходимостью преобразования каких-либо величин в другие по определенному закону. Закон может выражаться либо математической зависимостью между преобразуемыми величинами, либо в форме таблицы. Для второго варианта в ассемблере существует специальная команда xlat, которая осуществляет выборку байта из таблицы. В регистре BX должен находиться относительный адрес таблицы, в регистре AL — смещение в таблице к выбираемому байту. Выбранный из таблицы байт загружается в регистр AL, при этом содержимое AL перезаписывается. Размер таблицы может достигать 256 байт.

Следующий пример показывает преобразование беззнакового числа в регистре AL в символьное представление. Исходный код 16-разрядного приложения представлен в листинге 7.35.

**Листинг 7.35.** Преобразование числа в символьное представление командой xlat (16-разрядная версия)

```
  .model small
  .data
    tbl    DB "0123456789"   ; таблица преобразования
    res    DB 2 dup (' ')    ; область памяти для результата
           DB '$'
  .code
    start:
     mov    AX, @data
     mov    DS, AX
     lea    BX, tbl             ; загружаем адрес таблицы в регистр BX
     lea    SI, res             ; адрес результата -> SI
     mov    DX, SI              ; сохраняем адрес результата в регистре DX
     mov    AL, 19              ; десятичное число для преобразования -> AL
     cbw                        ; преобразуем байт в слово
     mov    CL, 10              ; делитель -> CL
     div    CL                  ; делим число 19 на 10
                                ; в результате: AL = 1, AH = 9 (остаток)
     xlat                       ; преобразовать AL в символ
     mov    byte ptr [SI], AL   ; сохранить в первом элементе строки
     xchg   AH, AL              ; обменять байты
     xlat                       ; преобразовать в символ
     mov    byte ptr [SI]+1, AL ; сохранить AL во втором элементе строки
```

продолжение ⇗

**Листинг 7.35** (продолжение)

```
    mov   AH, 9h          ; вывести результат на экран
    int   21h
    mov   AX, 4c00h
    int   21h
    end   start
    end
```

В этой программе десятичное число 19 преобразуется в строку символов 19 и выводится на экран. Для этого содержимое регистра AL, в котором находится число, преобразуется в двоично-десятичное число, а затем каждый разряд такого числа с помощью таблицы tbl конвертируется в символьное представление. Символы помещаются в соответствующем порядке в строку res, после чего выводятся на экран.

# 7.7. Полезные советы

Как видим, при манипуляциях со строками можно обходиться и без специализированных команд, однако из-за дополнительных операций инкремента-декремента адресов и дополнительного анализа условий равенства символов и конца строк код получается несколько громоздким. Самая высокая скорость выполнения строковых операций достигается обычно при копировании одной строки в другую или при перемещении элементов строки из одной области памяти в другую. Это особенно заметно при перемещении больших объемов данных.

Меньший выигрыш в производительности по сравнению с обычными командами дают команды поиска и сканирования. На скорость выполнения строковых операций влияет и размерность операндов. Вот несколько полезных советов для работы с командами строковых примитивов:

- При использовании цепочечных команд movs, stos, cmps и scas в 16-разрядных приложениях не забывайте инициализировать регистры DS и ES.
- Устанавливайте флаг направления DF в соответствии с направлением обработки строк или массивов.
- Не забывайте устанавливать в регистрах EDI (DI) и ESI (SI) необходимые значения. Например, команда movsw предполагает использование операндов DI и SI, а команда cmpsd — ESI и EDI.
- Инициализируйте регистр ECX (CX) в соответствии с количеством байтов или слов, участвующих в процессе обработки.
- Для обычной обработки нужно использовать префикс rep (команды movs и stos) и модифицированный префикс repe или repne (команды cmps и scas).
- Помните об обратной последовательности байтов в сравниваемых словах при выполнении команд cmpsw и scasw.
- При обработке справа налево устанавливайте начальные адреса на последний байт обрабатываемой области. Если, например, строка STRING1 имеет длину 16 байт, то для побайтовой обработки данных в этой области справа налево начальный адрес, загружаемый командой lea, должен быть STRING1 + 15. Для обработки слов начальный адрес в этом случае равен STRING1 + 14.

- При использовании команды lodsb необходимо учитывать то, что на процессорах Intel Pentium эта команда работает медленнее, чем блок команд

```
mov AL, byte ptr [ESI]
inc   ESI
```

Или такой блок:

```
mov AL, byte ptr [ESI]
add  ESI, 1
```

- То же самое касается и команд lodsw и lodsd. Для повышения производительности команду lodsw можно заменить блоком команд

```
mov AX, [SI]
add SI, 2
```

Аналогично, команду lodsd можно заменить таким блоком:

```
mov EAX, [ESI]
add ESI, 4
```

- При пересылке больших блоков байтов наиболее эффективно использование команды rep movsb. Для небольших блоков данных более эффективным является блок команд

```
mov AL, byte ptr [ESI]
inc ESI
mov byte ptr [EDI], AL
inc EDI
```

- То же самое касается команд rep movsw и rep movsd. В этом случае более высокую производительность при пересылке небольших блоков размером в слово обеспечивает следующий фрагмент программного кода:

```
mov AX, word ptr [SI]
add ESI, 2
mov word ptr [EDI], AX
add EDI, 2
```

А для двойных слов наиболее эффективен такой блок:

```
mov EAX, [ESI]
add ESI, 4
mov [EDI], EAX
add EDI, 4
```

- Команда rep scasb на процессорах Intel Pentium работает медленнее, чем следующий фрагмент программного кода:

```
next:
  mov AL, byte ptr [EDI]
  inc EDI
  cmp AL, reg2
  je  exit
  dec CX
  jnz next
exit:
```

- Команду stosb можно заменить комбинацией команд

  ```
  mov [DI]. AL
  inc DI
  ```

  А фрагмент программного кода для замены команды stosw выглядит так:

  ```
  mov [DI]. AX
  add DI. 2
  ```

  Наконец, команда stosd может быть реализована следующим образом:

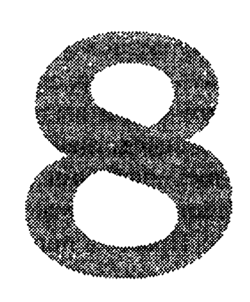
  ```
  mov [EDI]. EAX
  add EDI. 4
  ```

Заканчивая рассмотрение материала, хотелось бы сделать некоторые обобщения относительно использования цепочечных команд. Наиболее производительный код можно написать, если соблюдать несколько условий:

- данные в источнике и приемнике должны быть выровнены по 8-байтовой границе;
- флаг направления должен быть установлен в сторону увеличения адресов;
- счетчик в регистре ECX должен иметь значение, большее или, по крайней мере, равное 64;
- разность между содержимым регистров ESI и EDI должна быть больше или равна 32.

При разработке высокоэффективного кода желательно все же избегать цепочечных команд, используя вместо них комбинацию mov/inc или mov/dec.

# Арифметические и логические операции



Данная глава посвящена арифметическим и логическим операциям, выполняемым центральным процессором. К группе арифметических операций можно отнести:

- сложение;
- вычитание;
- умножение;
- деление;
- логический и циклический сдвиг.

Арифметические операции можно выполнять для беззнаковых и знаковых целочисленных данных.

К логическим операциям можно отнести несколько команд, манипулирующих отдельными битами операндов. В группу логических команд входят команды логического умножения (логическое И), логического сложения (логическое ИЛИ) и сложения по модулю 2 (исключающее ИЛИ).

В этой главе приводятся как описания арифметических и логических команд, так и примеры их применения. Кроме того, здесь же рассматриваются операции преобразования между двоичными данными и ASCII-кодами, которые играют существенную роль в математических вычислениях.

Наш анализ начнем с логических команд процессора Intel Pentium.

## 8.1. Логические команды

Первая команда, которую нам предстоит рассмотреть, — команда and. Эта команда осуществляет логическое (поразрядное) умножение первого операнда на второй. Исходное значение операнда-приемника при этом теряется и замещается

результатом умножения. В качестве первого операнда команды and можно указывать регистр, за исключением сегментного, или ячейку памяти. Вторым операндом может выступать регистр, за исключением сегментного, ячейка памяти или непосредственное значение. При этом не допускается, чтобы оба операнда являлись ячейками памяти. Операнды могут быть байтами, словами или двойными словами. Результат выполнения команды воздействует на флаги SF, ZF и PF.

Правила поразрядного умножения достаточно просты: если хотя бы один из попарно умножаемых битов является логическим нулем, то независимо от значения второго бита результат будет нулевым. Результирующий бит равен 1 только в том случае, когда соответствующие биты обоих операндов равны 1. Далее приводится простой пример, демонстрирующий логику работы операции логического умножения.

В этом примере op1 и op2 — 8-разрядные множители, а операнд op12_and — результат:

```
op1→01011100B
and
op2→11011011B
-----------------------------
op12_and→01011000B
```

Следующий пример показывает реализацию операции логического И на языке ассемблера:

```
mov AX.0FFEh
and AX.0Fh     ; после операции AX = 0Eh
```

В этом примере в качестве одного из операндов используется ячейка памяти

```
.data
  op dw 0A003h
.code
. . .
mov  AX. 100Eh
and  AX. op     ; после операции AX = 2h
```

Еще один пример демонстрирует использование 32-разрядных операндов:

```
mov EDX. 003A21A47h
and EDX. 0DB0100EFh  ; после операции EDX = 3000047h
```

Следующей командой, которая относится к группе логических команд и которую мы рассмотрим, является команда or (логическое ИЛИ).

Команда or выполняет операцию логического (поразрядного) сложения двух операндов. Результат замещает операнд-приемник, а операнд-источник не изменяется. В качестве первого операнда может выступать регистр (за исключением сегментного) или ячейка памяти, вторым операндом может служить регистр (кроме сегментного), ячейка памяти или непосредственное значение.

Не допускается определять оба операнда одновременно как ячейки памяти. Сам операнд может быть байтом, словом или двойным словом. Результат опе-

рации воздействует на флаги OF, SF, ZF, PF и CF, при этом флаги CF и OF всегда сбрасываются в 0.

Правила поразрядного сложения просты: если хотя бы один из попарно слагаемых битов является логической единицей, то независимо от значения второго бита результат равен единице. Результирующий бит равен 0 только в том случае, когда соответствующие биты обоих операндов равны 0. Далее приводится простой пример операции логического сложения.

В этом примере op1 и op2 — 8-разрядные слагаемые, а операнд op12_or — результат:

```
op1→01011100B
or
op2→11011011B
----------------------------
op12_or→11011111B
```

Следующий пример показывает реализацию операции логического ИЛИ на языке ассемблера:

```
mov AX. 000Fh
mov BX. 00F0h
or  AX. BX      ; после операции AX = 00FFh. BX = 00F0h
```

Как уже упоминалось, команда or допускает задание непосредственного значения в качестве второго операнда, что демонстрирует еще один пример:

```
mov AX.001Fh
or AX. 70A1h     ; после операции: AX = 70BFh
```

Использование командой or в качестве операнда ячейки памяти иллюстрирует следующий фрагмент программного кода:

```
.data
  msk db 97h
.code
. . .
mov CH. 0E1h
or CH. msk          ; после операции: CH = 0F7h
```

Допускается использование 32-разрядных операндов, как видно из примера:

```
.data
  op dd 8F000039h
.code
. . .
or op. 0A0h     ; после операции: op = 8F0000B9h
```

К группе логических команд относится и команда xor (исключающее ИЛИ), выполняющая операцию логического (поразрядного) исключающего ИЛИ над двумя операндами. Результат операции перезаписывает первый операнд, а второй операнд остается неизменным. Отдельные биты результата устанавливаются в 1, если соответствующие биты операндов различны, или в 0, если соответствующие биты операндов совпадают.

В качестве первого операнда команды xor допускается указывать регистр (за исключением сегментного) или ячейку памяти, в качестве второго — регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако нельзя определять оба операнда как ячейки памяти. Операнды могут быть байтами, словами или двойными словами. Команда воздействует на флаги OF, SF, ZF, PF и CF, причем флаги OF и CF всегда устанавливаются в 0, а состояние остальных флагов зависит от результата.

Далее приводится простой пример, демонстрирующий логику работы операции исключающего ИЛИ. В этом примере op1 и op2 — 8-разрядные множители, а операнд op12_xor — результат:

op1→01011100B
xor
op2→11011011B
------------------------------
op12_xor→10000111B

Несколько примеров демонстрируют технику применения команды xor с различными типами операндов.

В качестве одного из операндов может использоваться непосредственное значение:

```
mov AX.0ECh
xor AX.0F1FFh : после операции AX = 0F113h
```

Команда xor может применяться для обнуления регистра:

```
xor EDI, EDI  ; обнуление EDX
```

Оба операнда команды могут быть регистрами. Следующий фрагмент программного кода иллюстрирует этот случай:

```
mov AX, 0D5FEh
mov BX, 3Fh
xor AX, BX    : после операции AX = 0D5C1h, BX = 3Fh
```

Для 32-разрядных операндов допустим такой фрагмент программного кода:

```
mov EAX.0F734E1AAh
xor EAX.31DDCh : после операции EAX = 0F737FC76
```

Отдельную группу команд, которую условно можно отнести к логическим командам, составляют команды сканирования битов.

# 8.2. Команды сканирования битов

Группа команд сканирования битов служит для анализа отдельных битов операнда. Рассмотрим эти команды по порядку, начиная с команды bsf.

Команда bsf (Bit Scan Forward) сканирует слово или двойное слово в поисках бита, равного 1. Сканирование выполняется начиная с младшего бита по направлению к старшему. Если в операнде не обнаружены единичные биты, то устанавливается флаг ZF. Если единичные биты есть, то номер первого из них заносится

в указанный в команде регистр. Номер бита — это его позиция в операнде, причем самый младший бит имеет номер 0.

Первым операндом команды bsf должен быть регистр, в который помещается результат сканирования, вторым — регистр или ячейка памяти со сканируемым словом. Команда bsf допускает использование как 16-разрядных, так и 32-разрядных операндов (но как первый, так и второй операнд должен быть одного типа).

Примеры использования команды bsf:

```
mov BX,70h  ; помещаем в регистр BX значение 00000000 01110000B
bsf AX, BX  ; после выполнения команды AX = 4, ZF = 0
mov DX, 0   ; помещаем в регистр DX значение 0
bsf BX, DX  ; BX остался без изменения, флаг ZF = 1
mov DX, 8   ; в регистр DX помещается значение 00000000 00001000B
bsf BX, DX  ; после операции: BX = 3, ZF = 0
.data
  op DW 4000h  ; op = 01000000 00000000B
.code
. . .
bsf AX, op     ; после операции AX = 000Eh (14), ZF = 0
```

Команда bsr (Bit Scan Reverse) выполняет обратное сканирование битов слова или двойного слова в поисках бита, равного 1. Сканирование осуществляется по направлению от старшего бита к младшему. Если в операнде не обнаружены единичные биты, то устанавливается флаг ZF. Если единичные биты есть, то номер первого из них помещается в указанный в команде регистр. Номером бита считается его позиция в слове, отсчитываемая от бита 0. Первым операндом команды bsr нужно указывать регистр, в который помещается результат сканирования, вторым операндом может быть регистр или ячейка памяти со сканируемым словом. Команда bsr допускает использование 16-разрядных или 32-разрядных операндов, но оба операнда должны иметь один и тот же тип. Единственным исключением является случай, когда в качестве второго операнда выступает константа.

Рассмотрим несколько примеров использования команды bsr:

```
mov BX, 170h  ; в регистр BX помещаем 0000 0001 0111 0000B
bsr AX, BX    ; после выполнения операции AX = 8, ZF = 0
mov DX, 0     ; помещаем значение 0 в регистр DX
bsr BX, DX    ; после операции BX - без изменения, ZF = 1
mov DX, 8     ; помещаем в DX значение 0000 0000 0000 1000B
bsf BX, DX    ; после операции BX = 3, ZF = 1
.data
  op dw 5000h  ; op = 0101 0000 0000 0000B
.code
. . .
bsr AX, op     ; после операции AX = 000Eh (14), ZF = 0
```

Следующая команда, которую мы проанализируем, — команда bt (Bit Test). С ее помощью можно определить, установлен ли в заданном операнде определенный бит. Первым операндом команды является анализируемое значение, вторым — номер бита. В качестве первого операнда команды bt может выступать регистр или ячейка памяти, в качестве второго — регистр или непосредственное значение.

Команда допускает использование 16-разрядных или 32-разрядных операндов, но оба операнда должны иметь один и тот же тип. Исключением является случай, когда в роли второго операнда выступает константа. Полученное значение проверяемого бита становится значением флага CF.

Рассмотрим несколько примеров использования команды bt:

```
mov AX, 00FFh     ; помещаем значение 0000 0000 1111 1111B в регистр AX
bt AX, 7          ; после выполнения операции: бит 7 = 1, CF = 1
mov AX, 00FFh     ; помещаем значение 0000 0000 1111 1111B в регистр AX
bt AX,8           ; после выполнения операции: бит 8 = 0, CF = 0
mov AX, 9007h     ; помещаем значение 1001 0000 0000 0111B в регистр AX
mov BX, 12        ; помещаем номер проверяемого бита в регистр BX
bt AX, BX         ; после выполнения операции: бит 12 = 1, CF = 1
.data
 op dw 21h         ; в переменной op значение 0000 0000 0010 0001B
.code
. . .
bt op, 5          ; после выполнения операции: бит 5 = 1, CF = 1
```

Команда btc (Bit Test with Compliment) проверяет бит, указанный вторым операндом, в первом операнде, затем делает его значением флага CF и инвертирует. Таким образом, номер бита выступает вторым операндом, а первым может служить регистр или ячейка памяти. Кроме того, можно указать номер бита или непосредственно, или через регистр. Команда допускает использование как 16-разрядных, так и 32-разрядных операндов, но оба операнда должны быть одного типа. Исключением является случай, когда в качестве второго операнда выступает константа.

Рассмотрим примеры применения команды btc:

```
mov AX, 0076h  ; в регистр AX помещаем значение 0000 0000 0111 1100B
btc AX, 5      ; после операции AX = 0056h, CF = 1, бит 5 инвертирован в 0
mov AX, 00FFh  ; в регистр AX помещаем значение 0000 0000 1111 1111B
btc AX, 8      ; после операции AX = 01FFh, CF = 0, бит 8 инвертирован в 1
mov AX, 0E001h ; в регистр AX помещаем значение 1110 0000 0000 0001B
mov BX, 14     ; в регистр BX -> номер проверяемого бита
btc AX,BX      ; после операции AX = 0A001h, CF=1, бит 14 инвертирован в 0
.data
  op dw 0A7h   ; в переменной op значение 1010 1111B
.code
. . .
btc op, 5      ; после операции op = 87h, CF = 1, бит 5 инвертирован в 0
```

# 8.3. Команды сдвига и циклического сдвига

Команды сдвига и циклического сдвига предоставляют целый ряд возможностей по манипулированию данными байта, слова и двойного слова. Эти команды производят перемещение битов в операндах влево или вправо, причем сдвиг может быть как логическим (знак операнда не учитывается), так и арифметическим (для знаковых операндов).

Все команды сдвига и циклического сдвига используют флаги переноса CF и переполнения OF. При выполнении команд сдвига флаг CF всегда содержит значение последнего выдвинутого бита:

- shr — логический (беззнаковый) сдвиг вправо;
- shl — логический (беззнаковый) сдвиг влево;
- shld — логический сдвиг двойного слова влево;
- shrd — логический сдвиг двойного слова вправо;
- sal — арифметический сдвиг влево;
- sar — арифметический сдвиг вправо.

Циклический сдвиг представляет собой операцию сдвига, при которой выдвинутый бит занимает освободившийся разряд:

- ror — циклический сдвиг вправо;
- rol — циклический сдвиг влево;
- rcr — циклический сдвиг вправо с переносом;
- rcl — циклический сдвиг влево с переносом.

Вначале проанализируем, как работают команды сдвига. В следующем фрагменте показано использование команды shr:

```
mov  CL, 03              ; счетчик количества сдвигов -> CL
mov  BL, 10110111B       ; 10110111B -> BL
shr  BL, 1               ; после операции BL = 01011011B
shr  BL,CL               ; после операции BL = 00001011B
```

Первая команда shr сдвигает содержимое регистра BL вправо на 1 бит. Выдвинутый в результате бит становится значением флага CF, а самый левый бит регистра BL заполняется нулем. Вторая команда сдвигает содержимое регистра BL еще на три бита. При этом флаг CF последовательно принимает значения 1, 1, 0, а в три левых бита в регистре BL заносятся нули.

Посмотрим, как выполняется команда арифметического сдвига вправо sar:

```
mov  CL, 03           ; счетчик количества сдвигов -> CL
mov  AL, 10110111B    ; 10110111B -> AL
sar  AL, 1            ; после операции AL = 11011011B
sar  AL, CL           ; после операции AL = 11111011B
```

Команда sar отличается от команды shr тем, что для заполнения левого бита используется знаковый бит. Таким образом, положительные и отрицательные величины сохраняют свой знак. В приведенном примере знаковый бит содержит единицу.

При сдвигах влево правые биты заполняются нулями, поэтому команды сдвига shl и sal дают одинаковый результат.

Сдвиг влево часто используется для умножения числа на степень двойки, а сдвиг вправо — для деления на степень двойки. Такие операции выполняются значительно быстрее, чем обычные команды умножения или деления. При делении пополам нечетных чисел результатом становятся значения, округленные в меньшую

сторону. Например, деление чисел 5 или 7 на 2 дает результат 2 и 3 соответственно, и, кроме этого, флаг CF устанавливается в 1. Более того, при выполнении, например, сдвига на 2 бита более эффективным является использование двух команд сдвига, а не одной команды с предварительной загрузкой регистра CL значением 2. Для проверки бита переноса (флага CF) по окончании операции можно выполнить команду jc.

Перейдем к анализу команд циклического сдвига. Общей особенностью таких команд является то, что самые старшие (младшие) сдвигаемые биты переходят в самые младшие (старшие) позиции одного и того же операнда. Этим команды циклического сдвига отличаются от команд обычного сдвига, в которых выдвигаемые из операнда биты теряются. В циклическом сдвиге может быть задействован и флаг переноса CF, при этом выдвигаемый бит помещается в CF, а предыдущее значение флага переноса помещается в самый старший (младший) бит операнда.

Следующий фрагмент программного кода иллюстрирует операцию циклического сдвига ror:

```
mov  CL.2h          : счетчик сдвигов  -> CL
mov  AL.10011101B   : число 10011101B -> AL
ror  AL.1           : после операции AL = 11001110B
ror  AL.CL          : после операции AL = 10110011B
```

Первая команда ror при выполнении циклического сдвига перемещает правый единичный бит регистра AL в освободившуюся левую позицию. Вторая команда ror переносит, таким образом, два правых бита.

В командах rcr и rcl в сдвиге участвует флаг CF. Выдвигаемый из регистра бит заносится в флаг CF, а значение CF при этом поступает в освободившуюся позицию.

Далее приведен пример использования команды rcl:

```
mov  CL.2h          : счетчик сдвигов  -> CL
mov  AL.11011101B   : число 11011101B -> AL
rcl  AL.1           : после операции AL = 11001110B
rcl  AL.CL          : после операции AL = 10110011B
```

Рассмотрим пример, в котором используются команды циклического и простого сдвигов. Предположим, что 32-разрядное значение находится в регистрах DX:AX, так что левые 16 бит располагаются в регистре DX, а правые — в AX. Для умножения на 2 этого значения допустимы две следующие команды:

```
shl  AX.1     : Умножение пары регистров
rcl  DX.1     : DX:AX на 2
```

Здесь команда shl сдвигает все биты регистра AX влево, причем самый левый бит становится значением флага CF. Затем команда rcl сдвигает все биты регистра DX влево, и в освободившийся правый бит заносит значение из флага CF.

# 8.4. Обработка целых чисел

Вначале кратко рассмотрим некоторые теоретические аспекты обработки числовых данных. Процессор Intel Pentium имеет команды для обработки целочисленных арифметических данных двух форматов: двоичного и двоично-десятичного

(BCD). Данные в двоичном формате могут интерпретироваться как числа со знаком или без знака, при этом не существует отдельных форматов для представления знаковых и беззнаковых чисел. Одно и то же двоичное представление может рассматриваться и как значение со знаком, и как значение без знака в зависимости от того, как трактуется старший бит операнда. Для чисел без знака он является старшим значащим битом операнда. В то же время для чисел со знаком нулевое значение соответствует положительному значению, а единичное — отрицательному числу. Остальные разряды операнда — значащие.

Следует заметить, что значащие разряды не всегда представляют абсолютное значение (модуль) числа. Это верно только для положительных чисел, а для отрицательного числа они представляют так называемый дополнительный код числа.

Двоично-десятичные числа, в свою очередь, могут быть представлены в одном из следующих форматов:

- в формате ASCII;
- неупакованном двоично-десятичном формате;
- упакованном двоично-десятичном формате.

Рассмотрим эти форматы более подробно. Представление чисел в формате ASCII удобно в тех случаях, когда необходимо выполнять ввод чисел с консоли или вывод на какое-либо устройство, например дисплей или принтер. Для получения правильного результата при выполнении арифметических операций с такими числами необходимо проводить корректировку результата с помощью специальных команд. Операции с числами в формате ASCII будут рассмотрены более подробно далее. ASCII-числа представлены следующим образом: старший (левый) полубайт каждого байта содержит значение 3h, а младший (правый) полубайт — значение десятичного разряда.

Например, число 6591 в формате ASCII представлено как 36353931h, при этом самый старший байт содержит значение 36h, а самый младший — 31h.

Числа в неупакованном двоично-десятичном формате отличаются от их ASCII-представления тем, что левые полубайты таких чисел установлены в 0. Это значительно облегчает выполнение операций умножения и деления над такими числами. Например, число 6591 в неупакованном формате выглядит как 06050901h, причем самый старший байт содержит число 06h, а самый младший — 01h.

Следует заметить, что операции над неупакованными двоично-десятичными числами выполняются медленнее, чем над двоичными. Это связано с тем, что неупакованные числа обрабатываются байт за байтом. Преимуществом неупакованных чисел является то, что с их помощью можно легко организовать программную обработку больших чисел.

Последний формат представления чисел из списка — упакованный двоично-десятичный формат. Каждый байт упакованного числа содержит две десятичные цифры, то есть каждое десятичное число представлено четырьмя битами. Например, число 6591 в упакованном формате будет представлено как 6591h, причем старший байт содержит значение 65h, а младший — 91h. Упакованные двоичные числа можно только складывать и вычитать, другие операции требуют дополнительного преобразования в неупакованный формат. В настоящее время упакованные двоично-десятичные числа находят ограниченное применение.

Остановимся на особенностях манипуляций знаковыми и беззнаковыми числами. Аппаратная интерпретация процессором старшего бита операнда реализована в командах imul и idiv. Команды add и sub не делают разницы между знаковыми и беззнаковыми величинами, они просто складывают и вычитают биты, поэтому в этих случаях забота о правильной трактовке старшего бита ложится на программное обеспечение. Хочу особо подчеркнуть, что процессор ничего не предполагает относительно знака числа и выполняет вычисления двоичных значений. При этом фиксируется ситуация выхода за пределы разрядной сетки операнда (флаг CF) и состояние старшего разряда (флаг OF).

Еще одной особенностью, которую следует учитывать при выполнении арифметических операций, является переполнение. Причина этого — ограниченность разрядной сетки операндов. При выполнении операции сложения или умножения возможен выход результата за пределы разрядной сетки. Если результат больше максимально представимого значения для операнда данной размерности, то говорят о ситуации переполнения. Иначе, если результат меньше минимально представимого числа, то говорят о ситуации антипереполнения.

Рассмотрим более подробно операции над целыми числами и начнем с операций сложения и вычитания.

Команды add и sub выполняют сложение и вычитание байтов или слов, содержащих двоичные данные. Вычитание выполняется в компьютере по методу сложения с двоичным дополнением: для второго операнда устанавливаются обратные значения битов и прибавляется 1, затем выполняется операция сложения с первым операндом. Во всем, кроме первого шага, операции сложения и вычитания идентичны.

В зависимости от размера и типа операндов, участвующих в операциях сложения и вычитания, возможны следующие ситуации:

- сложение/вычитание регистр-регистр;
- сложение/вычитание память-регистр;
- сложение/вычитание регистр-память;
- сложение/вычитание регистр-непосредственное значение;
- сложение/вычитание память-непосредственное значение.

При выполнении операций сложения/вычитания, как и для других арифметических операций, для получения правильного результата необходимо контролировать состояние некоторых флагов процессора. Это касается флагов CF и OF. С помощью этих флагов программа должна учитывать возможное переполнение результата и перенос в старшие разряды. Для этого отслеживаются условия, задаваемые флагами, и выполняются действия:

- CF = OF = 0 — результат правильный и является положительным числом;
- CF = 1, OF = 0 — результат правильный и является отрицательным числом;
- CF = OF = 1 — результат неправильный и является положительным числом, хотя правильный результат должен быть отрицательным (для корректировки необходимо увеличить размер результата в два раза и заполнить это расширение нулевым значением);

- CF = 0, OF = 1 — результат неправильный и является отрицательным числом, хотя правильный результат должен быть положительным (для корректировки необходимо увеличить размер результата в два раза и произвести расширение знака).

Выполнение арифметических операций иногда может приводить к ситуации переполнения. Рассмотрим, например, операцию сложения для знаковых операндов размерностью в 1 байт. Один байт содержит знаковый бит и 7 бит данных, то есть диапазон допустимых значений находится между −128 и +127. Не исключена возможность, что результат арифметической операции может легко превзойти емкость однобайтового регистра.

Необходимо учитывать и то, что результат сложения в регистре AL, превышающий его емкость, автоматически не переходит в регистр AH. Предположим, что регистр AL содержит 60h, тогда после выполнения следующей команды в AL будет находиться значение 80h:

```
add AL, 20h
```

Кроме того, устанавливаются флаг переполнения и знаковый флаг. Причина заключается в том, что шестнадцатеричное значение 80 (двоичное 10 000 000) является отрицательным числом. Таким образом, вместо +128 мы получаем −128. Очевидно, что размерность регистра AL недостаточна для такой операции, поэтому можно использовать регистр AX. Увеличить размерность операнда можно с помощью команды cbw (Convert Byte to Word — преобразовать байт в слово).

В следующем примере значение 60h в регистре AL преобразуется в шестнадцатеричное значение 60 в регистре AX. Знаковый бит передается в регистре AH. В этом случае команда add дает правильный результат и в регистре AX будет находиться значение, равное шестнадцатеричному значению 0080 или в десятичной нотации +128:

```
cbw          ; расширить AL до AX
add AX, 20H  ; прибавить 20H к AX
```

Следует заметить, что полное 16-разрядное слово имеет также ограничение: один знаковый бит и 15 бит данных, что соответствует значениям от −32 768 до +32 767.

Лучше всего анализировать эти операции на практических примерах. В первом примере выполняется сложение двоичных чисел размером в 1 байт без учета знака. Программный код реализован в виде процедуры addb_unsigned и показан в листинге 8.1.

В этом примере результат сложения сохраняется в переменной sum, а возможное переполнение из-за недостаточной размерности операндов фиксируется в переменной carry. Таким образом, программа учитывает возможное переполнение результата. Например, если op1 содержит значение 140, а op2 — 119, то после сложения в переменной sum будет содержаться значение 3, а переменная carry получит значение 1. Это легко объяснимо, поскольку произошло переполнение регистра AL — результат превысил значение 256.

**Листинг 8.1.** Сложение однобайтовых чисел без знака

```
. . .
.data
op1     db ?      ; первый операнд
op2     db ?      ; второй операнд
carry   db 0      ; здесь запоминается флаг переноса
sum     db 0      ; здесь хранится результат сложения
.code
addb_unsigned proc
  clc             ; очистка флага переноса перед сложением
  mov AL, op1
  add AL, op2
  jnc exit        ; проверка на переполнение
  add carry,1     ; сохраняем флаг переноса
exit:
  mov  sum, AL
  ret
addb_unsigned endp
```

При вычитании однобайтовых чисел вместо команды add применяется команда sub. Кроме того, операция вычитания может дать отрицательное число, и это необходимо учитывать для правильной интерпретации результата. В следующем примере выполняется вычитание двоичных чисел размером в 1 байт. Программный код реализован в виде процедуры sub_bytes (листинг 8.2).

**Листинг 8.2.** Вычитание однобайтовых чисел

```
. . .
.data
op1          db ?      ; первый операнд
op2          db ?      ; второй операнд
carry        db 0      ; здесь запоминается флаг переноса
substract    db 0      ; здесь хранится результат вычитания
.code
sub_bytes proc
  clc             ; очистка флага переноса перед сложением
  mov AL, op1
  sub AL, op2
  jnc exit        ; проверка на переполнение
  add carry,1     ; сохраняем флаг переноса
exit:
  mov  substract, AL
  ret
sub_bytes endp
```

По сравнению с предыдущим примером здесь все команды сложения заменены аналогичными командами вычитания. Для значений op1 и op2, равных, например, 119 и 140 соответственно, после операции вычитания переменная substract будет содержать шестнадцатеричное значение EB. Это значение можно рассматривать и как беззнаковое число 235, и как отрицательное –21. По смыслу задачи разность op1 и op2 должна быть равной –21. Из этого примера видно, что интерпретация результата арифметической операции возлагается на программиста.

Хочу сделать важное замечание. Поскольку результат выполнения операции — отрицательное число, дополнительно устанавливается флаг знака SF. Этот флаг

можно использовать для анализа результата арифметической операции в процессе разработки программного кода.

Вернемся к первому примеру (см. листинг 8.1). Видно, что сохраненное в переменной sum значение равно 3, в то время как правильный результат должен быть 259. Для того чтобы получить реальное значение sum, необходимо сделать некоторые изменения в программе. Модифицированный вариант программы показан в листинг 8.3.

**Листинг 8.3.** Модифицированный вариант сложения однобайтовых чисел

```
. . .
.data
  opl   DB 140
  op2   DB 119
  sum   DW 0
.code
addb_unsigned proc
  xor AX. AX
  clc
  mov AL. opl
  adc AL. op2
  jnc exit
  adc  AH. 0
exit:
  mov  sum. AX
addb_unsigned endp
```

Проанализируем внесенные изменения. Во-первых, переменная sum имеет теперь разрядность слова, что позволяет расширить диапазон сохраняемых значений до 65 536. Во-вторых, при возникновении переноса он учитывается в старшем байте регистра AX (команда adc ah, 0). Наконец, сам результат имеет разрядность 16 бит, что в данном случае дает правильный результат — переменная sum будет содержать значение 259.

Сложение двоичных чисел большей размерности (2–4 байта) выполняется аналогично. Для этого необходимо заменить директиву DB на DW/DD и регистр AL на AX/EAX. Следующий пример демонстрирует это (листинг 8.4).

**Листинг 8.4.** Сложение двух чисел размером в слово

```
. . .
.data
opl     dw 1407    : первый операнд
op2     dw 9119    ; второй операнд
carry   db 0       : здесь запоминается флаг переноса
sum     dw 0       ; здесь хранится результат сложения
.code
addw_unsigned proc
  clc              ; очистка флага переноса перед сложением
  mov AX. opl
  add AX. op2
  jnc exit         ; проверка на переполнение
  add carry.1      : сохраняем флаг переноса
exit:
  mov  sum. AX
  ret
addw_unsigned endp
```

Результатом сложения двух слов, op1 и op2, является число 10 526. Операцию сложения двух слов выполняет процедура addw_unsigned. В большинстве современных программных продуктов приходится иметь дело с большими числами, представленными несколькими байтами. Например, для сложения многобайтовых чисел без знака требуется более сложный алгоритм, чем для байтов или слов. Чтобы понять принцип нахождения суммы многобайтовых чисел без знака, рассмотрим пример сложения двухбайтовых чисел и расширим наш алгоритм для случая произвольного числа байтов. Следующая процедура (назовем ее add_multibytes) складывает два двухбайтовых числа (листинг 8.5).

**Листинг 8.5.** Сложение двух двухбайтовых чисел

```
   .data
  op1  DW 3501
  op2  DW 781
  sum  DW 0
.code
add_multibytes proc
  clc
  xor AX, AX
  mov AL, byte ptr op1
  add AL, byte ptr op2
  mov byte ptr sum, AL
  mov AL, byte ptr op1+1
  adc AL, byte ptr op2+1
  mov byte ptr sum+1, AL
  ret
add_multibytes endp
```

Нахождение суммы операндов op1 и op2 выполняется по такой схеме: вначале находим сумму младших байтов этих операндов и заносим ее в младший байт переменной sum, которая будет содержать результат сложения. После этого находим сумму старших байтов переменных op1 и op2 и помещаем ее в старший байт переменной sum, при этом учитывается флаг переноса (вместо команды add применяется adc). Следует заметить, что размерность операндов op1 и op2 должна быть одинаковой. Изменим программный код процедуры add_multibytes (см. листинг 8.5) так, как показано в листинге 8.6.

**Листинг 8.6.** Сложение двухбайтовых чисел (улучшенная версия)

```
.data
  op1  DW 11901
  len  EQU $-op1
  op2  DW 5598
  sum  DW 0
.code
add_multibytes proc
  clc
  xor  AX, AX
  mov  CX, len
  lea  SI, byte ptr op1
  lea  DI, byte ptr op2
  lea  BX, byte ptr sum
```

```
next_byte:
  mov  AL, [SI]
  adc  AL, [DI]
  mov  byte ptr [BX], AL
  inc  SI
  inc  DI
  inc  BX
  loop next_byte
  ret
add_multibytes endp
```

Модифицированная процедура работает точно так же, как исходная, основное различие в том, что вычисление частичных сумм одинаковых байтов выполняется в цикле. При этом в счетчике CX содержится размер операндов в байтах. Эту процедуру можно использовать для суммирования большего числа байтов, если изменить значение в счетчике CX. Для заданных значений операндов op1 и op2 результат равен 17 499.

Вычитание многобайтовых чисел проводится по такой же схеме, с той лишь разницей, что команды сложения заменены командами вычитания. В листинге 8.7 показан пример побайтового вычитания двух слов с помощью процедуры sub_multibytes.

**Листинг 8.7.** Вычитание двухбайтовых чисел

```
. . .
.data
  op1        DW  1901
  len        EQU $-op1
  op2        DW  5598
  substract  DW  0
.code
sub_multibytes proc
  clc
  xor  AX, AX
  mov  CX, len
  lea  SI, byte ptr op1
  lea  DI, byte ptr op2
  lea  BX, byte ptr substract
next_byte:
  mov  AL, [SI]
  sbb  AL, [DI]
  mov  byte ptr [BX], AL
  inc  SI
  inc  DI
  inc  BX
  loop next_byte
  ret
sub_multibytes endp
```

Для данных значений операндов результат вычитания равен 3697.

Сложение/вычитание многобайтовых чисел по байтам или по словам позволяет выполнять арифметические операции над числами произвольной большой размерности (естественно, что диапазон таких чисел определяется архитектурой

компьютера). Приведу пример законченной процедуры на ассемблере, позволяющей находить сумму восьмибайтовых целых чисел. Процедура называется _add_8bytes, и ее исходный текст представлен в листинге 8.8.

**Листинг 8.8.** Сложение 8-байтовых чисел (32-разрядная версия)

```
.686
.model flat
option casemap:none
.data
  op1    DQ  15751
  len    EQU $-op1
  op2    DQ  91839
  sum    DQ  0
.code
 _add_8bytes proc
  push EBX
  clc
  xor  EAX, EAX
  mov  ECX, len
  lea  ESI, byte ptr op1
  lea  EDI, byte ptr op2
  lea  EBX, byte ptr sum
next_byte:
  mov  AL, [ESI]
  adc  AL, [EDI]
  mov  byte ptr [EBX], AL
  inc  ESI
  inc  EDI
  inc  EBX
  loop next_byte
  lea  EAX, sum
  pop  EBX
  ret
 _add_8bytes endp
 end
```

В основе работы алгоритма побайтового сложения лежит попарное сложение байтов, находящихся на одинаковых позициях в операндах-слагаемых. При этом необходимо учитывать флаг переноса, который может устанавливаться после сложения байтов. Хочу напомнить, что оба операнда должны иметь одинаковую размерность.

Ввиду наличия цикла используется только одна команда сложения adc. Перед выполнением цикла команда clc устанавливает нулевое значение флага переноса CF. Для того чтобы подобный алгоритм работал, необходимо обеспечить смежность слов, выполняя обработку справа налево. Кроме того, дополнительно установите счетчик байтов в регистре ECX.

Результатом выполнения программного кода при данных значениях операндов является число 107 590, помещенное в переменную sum. Эту процедуру можно использовать при разработке программы на одном из языков высокого уровня. Наиболее удобный способ сделать это — поместить 32-разрядный адрес переменной sum в регистр EAX и вернуть управление основной программе.

Например, программный код консольного 32-разрядного приложения в Visual C++ .NET может выглядеть так, как показано в листинге 8.9.

**Листинг 8.9.** Демонстрационная программа для процедуры из листинга 8.8

```
#include <stdio.h>
extern "C" unsigned int* add_8bytes(void);
int main(void)
{
  unsigned int* i1 = add_8bytes();
  printf("Result of adding 8 bytes = %u\n", *i1);
  getchar();
  return 0;
}
```

Во всех рассмотренных примерах размерность операндов, вовлеченных в операции сложения/вычитания, была одинаковой. Но что делать, если размеры операндов различны? Если операнды, участвующие в операции, предполагаются беззнаковыми, то проблему можно решить довольно просто: сформировать из данного операнда новый, повышенной размерности, за счет заполнения старших байтов сформированного операнда нулями. Например, если необходимо увеличить размерность беззнакового числа на 1 байт, то можно сформировать из него слово и заполнить старший байт нулями, оставив при этом младший байт без изменения. Проиллюстрировать вышесказанное можно на примере:

```
 . . .
.data
  op_byte DB 5
  op_word DW 0
.code
  . . .
  mov AL, op_byte
  mov byte ptr op_word, AL
  . . .
```

После выполнения этого фрагмента программного кода переменная op_word, имеющая размерность слова, в старшем байте будет содержать нули, а в младшем — значение 5.

Если необходимо преобразовать знаковое число, требуется более сложная процедура. Для решения подобных задач в процессор Intel Pentium включен ряд специальных команд, облегчающих процесс преобразования:

- cbw (Convert Byte to Word — преобразование байта в слово) — команда заполняет регистр AH знаковым битом числа, находящегося в регистре AL, что дает возможность выполнять арифметические операции над исходным операндом-байтом, как над словом в регистре AX. Команда не имеет параметров и не воздействует на флаги процессора;
- cwd (Convert Word to Double — преобразование слова в двойное слово) — команда преобразует слово в регистре AX в двойное слово в регистрах DX:AX, при этом старший бит в регистре AX распространяется на все биты регистра DX;

- cdq (Convert Double Word to Quarter Word — преобразование двойного слова в учетверенное) — двойное слово, находящееся в регистре EAX, преобразуется в учетверенное слово в регистрах EDX:EAX, при этом старший бит регистра EAX распространяется на все биты регистра EDX.

Команда cdq расширяет знак двойного слова в регистре EAX на регистр EDX. Эту команду можно использовать для образования четырехсловного делимого из двухсловного перед операцией двухсловного деления. Команда не имеет параметров и не воздействует на флаги процессора.

Для демонстрации работы команд преобразования типов рассмотрим пример, в котором вычитаются многобайтовые числа разного размера. Сама операция реализуется в процедуре _sub_8bytes (листинг 8.10).

**Листинг 8.10.** Вычитание многобайтовых чисел разного размера (32-разрядная версия)

```
.686
.model flat
option casemap:none
.data
  op1          DQ  15751
  len          EQU $-op1
  op2_dd       DD  97106
  op2          DQ  0
  substract    DQ  0
.code
 _sub_8bytes proc
  push EBX
  mov  ECX, len
  mov EAX, op2_dd          ; здесь выполняется преобразование двойного
                           ; слова op2_dd в учетверенное слово в op2
                           ; с помощью команды cdq
  cdq
  mov dword ptr op2, EAX
  mov dword ptr op2+4, EDX
  xor  EAX, EAX
  clc
  lea  ESI, byte ptr op1
  lea  EDI, byte ptr op2
  lea  EBX, byte ptr substract
next_byte:
  mov  AL, [ESI]
  sbb  AL, [EDI]
  mov  byte ptr [EBX], AL
  inc  ESI
  inc  EDI
  inc  EBX
  loop next_byte
  lea  EAX, substract
  pop  EBX
  ret
 _sub_8bytes endp
 end
```

Процедура _sub_8bytes посредством регистра EAX возвращает адрес переменной substract, содержащей результат вычитания.

Операция умножения для беззнаковых данных выполняется с помощью команды mul, а для знаковых — imul (Integer Multiplication — умножение целых чисел). Формат обрабатываемых чисел и выбор подходящей команды умножения определяет сам программист. Существует несколько форматов для команд умножения:

- Множимое находится в регистре AL, а множитель — в ячейке памяти размером в 1 байт или в однобайтовом регистре. После умножения результат помещается в регистр AX. Операция перезаписывает данные в регистре AH.
- Множимое находится в регистре AX, а множитель — в однословной ячейке памяти или в регистре. Произведение представляет собой двойное слово, старшая часть которого размещается в регистре DX, а младшая — в регистре AX. Операция перезаписывает данные, которые до этого находились в регистре DX.
- Множимое находится в регистре EAX, а множитель — в двухсловной ячейке памяти или в регистре. Произведение представляет собой два двойных слова, при этом старшее слово размещается в регистре EDX, а младшее — в регистре EAX. Операция перезаписывает данные, которые до этого находились в регистре EDX.

Команды mul и imul имеют единственный операнд, являющийся множителем. Проанализируем следующую команду:

```
mul opr
```

Здесь множителем является переменная opr. Если переменная opr определена как байт, то операция предполагает умножение содержимого AL на значение байта в переменной opr. Если переменная определена как слово, то операция предполагает умножение содержимого AX на значение слова, содержащегося в opr. Наконец, если переменная opr определена как двойное слово, то операция предполагает умножение содержимого EAX на значение двойного слова в переменной opr.

Если множитель находится в регистре, то размерность регистра определяет тип операции, например:

```
mul CL
```

Поскольку регистр CL содержит один байт, то в качестве множимого будет выбран регистр AL, а произведение помещается в регистр AX. Если выполняется следующая команда, то множитель в регистре BX имеет размерность слова, поэтому в качестве множимого выбирается регистр AX, при этом произведение помещается в пару регистров DX:AX:

```
mul BX
```

После выполнения команд mul и imul флаги CF и OF устанавливаются в том случае, если старший байт результата содержит значение, например:

```
mov  AL,37
mov  BL, 5
imul BL
```

После выполнения операции умножения регистр AX содержит 00B9h (+185), при этом CF = 1 и OF = 1. Регистр AH содержит значение (в данном случае — все нули). Рассмотрим другой пример:

```
mov  AL.-37
mov  BL. 5
imul BL
```

После выполнения операции умножения регистр AX содержит 0FF47h (–185). Поскольку в регистре AH содержится расширение знака регистра AL (0FFh), то флаги имеют следующие значения: CF = 0, OF = 0.

При выполнении операций умножения и деления (впрочем, это касается также и сложения/вычитания) нужно по возможности выбирать большую разрядность операндов. Во многих случаях это позволяет избежать ситуаций переполнения и вызываемых такими ситуациями ошибок.

Рассмотрим практические примеры использования команд mul и imul. Вначале проанализируем операции беззнакового умножения. Наш следующий пример состоит из двух процедур — _mul_words и _mul_mixed. С помощью процедуры _mul_words выполняется умножение двух операндов, op1_word и op2_word, имеющих размерность слова, а результат помещается в переменную op1_mul_op2 размерностью в двойное слово. Процедура возвращает в качестве результата адрес переменной op1_mul_op2 в регистре EAX.

Процедура _mul_mixed выполняет умножение операнда op_byte размерностью в 1 байт на op_word, имеющий размерность слова, и результат помещается в двухсловную переменную op2_mul_mixed. Процедура возвращает в качестве результата адрес переменной op_mul_mixed в регистре EAX. Программный код процедур представлен в листинге 8.11.

**Листинг 8.11.** Умножение чисел разной размерности (32-разрядная версия)

```
.686
.model flat
option casemap: none
.data
  op1_word     DW 37
  op2_word     DW 24
  op1_mul_op2  DD 0
  op_byte      DB 34
  op_word      DW 198
  op_mul_mixed DD 0
 .code
 _mul_words proc
   mov AX. op1_word
   mov BX. op2_word
   mul BX
   mov word ptr op1_mul_op2. AX
   mov word ptr op1_mul_op2+2. DX
   lea EAX. op1_mul_op2
   ret
  _mul_words endp
 _mul_mixed proc
```

```
  movzx AX, op_byte
  mov   BX, op_word
  mul   BX
  mov   word ptr op_mul_mixed, AX
  mov   word ptr op_mul_mixed+2, DX
  lea   EAX, op_mul_mixed
  ret
 _mul_mixed endp
 end
```

В процедуре _mul_mixed для расширения размерности операнда op_byte используется команда movzx (Move with Zero-Extend — копирование с расширением нуля). Результаты выполнения процедур равны 888 и 6732 для _mul_words и _mul_mixed соответственно.

При знаковом умножении используется команда imul. В следующем примере демонстрируется умножение однобайтового отрицательного операнда на двойное слово, реализованное в процедуре _imul_mixed (листинг 8.12).

**Листинг 8.12.** Умножение знаковых чисел (32-разрядная версия)

```
.686
.model flat
option casemap: none
.data
  op_byte       DB -31
  op_dword      DD 750
  op_imul_mixed DQ 0
 .code
  _imul_mixed proc
    movsx  AX, op_byte
    movsx  EAX, AX
    mov    EBX, op_dword
    imul   EBX
    mov    dword ptr op_imul_mixed, EAX
    mov    dword ptr op_imul_mixed+4, EDX
    lea    EAX, op_imul_mixed
    ret
 _imul_mixed endp
  end
```

В этом примере задействованы два операнда: op_byte размером в 1 байт и двухсловная переменная op_dword. Результат операции помещается в 8-байтовую переменную op_imul_mixed и при указанных значениях операндов равен –23 250. Перед выполнением умножения необходимо расширить размерность операнда op_byte до двойного слова. Для этого используются две команды movsx (Move with Sign Extend — копирование с расширением знака).

Первая команда movsx помещает 8-байтовый операнд в 16-разрядный регистр AX, расширяя знак на старшую половину AX (регистр AH). Вторая команда movsx преобразует 16-разрядное значение в 32-разрядное и помещает его в регистр EAX. Обратите внимание на то, что используется именно команда movsx, а не movzx! Команда movzx применяется только для беззнаковых операндов или в случаях, когда знак операнда не имеет значения.

Хочу заметить, что если множимое и множитель имеют одинаковый знаковый бит, то команды mul и imul генерируют одинаковый результат. Но если сомножители имеют разные знаковые биты, то результатом выполнения команды mul будет положительное число, в то время как команда imul даст отрицательное значение.

При решении практических задач очень часто требуется умножать операнды, состоящие из нескольких байтов или слов. Это позволяет находить произведение очень больших чисел, что требуется в различных задачах экономики и математики.

В этом случае операция умножения может выполняться для операндов размером в байт или слово. Хочу напомнить, что максимальное знаковое значение слова не может превышать +32 767, поэтому умножение больших чисел требует некоторых дополнительных операций. Один из простых вариантов умножения больших чисел предполагает попарное умножение отдельных слов и сложение полученных результатов. Алгоритм этой процедуры напоминает умножение столбиком при нахождении произведения десятичных чисел.

Далее рассмотрим пример умножения двух беззнаковых чисел, каждое из которых представлено двойным словом. Результат произведения сохраняется в 64-разрядной переменной. Программный код процедуры (назовем ее _mul_multibytes) представлен в листинге 8.13.

**Листинг 8.13.** Умножение беззнаковых чисел размером в двойное слово (32-разрядная версия)

```
.686
.model flat
option casemap:none
.data
  op1  DD 32267  ; |младшее слово = b |старшее слово = a |
  op2  DD 17904  ; |младшее слово = d |старшее слово = c |
  res  DQ 0
.code
 _mul_multibytes proc
   lea  ESI, op1      ; помещаем адрес op1 в ESI
   lea  EDI, op2      ; помещаем адрес op2 в EDI
                      ; вычисляем частичное произведение b * d
   clc
   mov   AX, word ptr [ESI] ; помещаем b в AX
   push  EAX                ; сохраняем EAX
   mov   BX, word ptr [EDI] ; помещаем d в BX
   mul   BX                 ; умножение AX * BX
   mov   word ptr res, AX   ; сохраняем младшую часть результата
   mov   word ptr res+2, DX ; сохраняем старшую часть результата
   pop   EAX                ; извлекаем EAX для вычисления второго произведения
                            ;   вычисляем частичное произведение b * c
   mov   BX, word ptr [EDI]+2 ; помещаем c в BX
   mul   BX                   ; умножение AX * BX
   add   word ptr res+2, AX   ;сложение частичных произведений b * d и b * c
   adc   word ptr res+4, DX   ; учесть перенос
   jnc   next                 ; возник перенос?
   inc   word ptr res+6       ; да, нужно его учесть в старшем слове
 next:
                              ;   вычисляем частичное произведение a * d
```

```
  mov   AX. word ptr [ESI]+2 ; помещаем a в AX
  push  EAX                  ; сохраняем EAX
  mov   BX. word ptr [EDI]   ; помещаем d  в BX
  mul   BX                   ; умножение AX * BX
  add   word ptr res+2. AX   ; прибавляем a * d к полному произведению
  adc   word ptr res+4. DX   ; учесть перенос
  pop   EAX                  ; извлекаем EAX для вычисления четвертого произведения
  jnc   hi                   ; возник перенос
  inc   word ptr res+6       ; да. нужно его учесть в старшем слове
hi:
                             ; вычисляем частичное произведение a * c
  mov   BX. word ptr [EDI]+2 ; помещаем c в BX
  mul   BX                   ; умножение AX * BX
  add   word ptr res+4. AX   ; прибавляем a * c к полному произведению
  adc   word ptr res+6. DX   ; учесть перенос
  lea   EAX. res             ; поместить в EAX адрес произведения
  ret
 _mul_multibytes endp
 end
```

Проанализируем алгоритм работы процедуры. Умножение операндов op1 и op2 выполняется пословно. Обозначим составные части двойных слов Op1 и op2 следующим образом:

- *b* — младшее слово op1;
- *a* — старшее слово op1;
- *d* — младшее слово op2;
- *c* — старшее слово op2.

Для нахождения произведения в целом вычисляются произведения 16-разрядных множителей $b \times d$, $b \times c$, $a \times d$, $a \times c$ и находятся частичные суммы этих произведений с учетом их позиции в полном произведении. При заданных значениях операндов произведение равно 577 708 368.

Эту процедуру можно задействовать при вычислении произведения беззнаковых целых чисел в программах на языках высокого уровня. Например, программный код простейшего 32-разрядного приложения на Visual C++, использующего процедуру _mul_multibytes, мог бы выглядеть так, как показано в листинге 8.14.

**Листинг 8.14.** Демонстрационная программа для процедуры из листинга 8.13

```
#include <stdio.h>
extern "C" int* mul_multibytes(void):
int main(void)
{
  printf("Result MULTIBYTE MUL %u\n". *mul_multibytes()):
  return 0:
}
```

Эффективность операций умножения можно повысить, если использовать несколько простых приемов. Например, при умножении на степень числа 2 (2, 4, 8 и т. д.) эффективнее вместо умножения выполнять логический сдвиг влево на требуемое число битов. Сдвиг более чем на 1 требует загрузки величины сдвига в регистр CL.

Рассмотрим несколько примеров. Предположим, что множимое находится в одном из регистров AL или AX. Тогда для умножения на 2 содержимого AL можно использовать команду

```
shl  AL.1
```

Для умножения содержимого AX на 8 можно воспользоваться такими командами:

```
mov  CL. 3
shl  AX. CL
```

Рассмотрим, как выполняется операция деления в процессорах Intel. Для деления беззнаковых данных используется команда div, а для знаковых — idiv. Какую из этих команд выбрать, решает разработчик программы. В зависимости от размерности операндов существуют следующие форматы операции деления:

- Деление слова на байт. Делимое находится в регистре AX, а делитель — в байте памяти или в однобайтовом регистре. После деления остаток помещается в регистр AH, а частное — в AL. Операция с данными типами операндов имеет ограниченное применение из-за небольшого диапазона допустимых значений (однобайтовое частное не превышает +255 для беззнакового деления и +127 — для знакового).
- Деление двойного слова на слово. Делимое находится в регистровой паре DX:AX, а делитель — в слове памяти или в регистре. После деления остаток помещается в регистр DX, а частное — в регистр AX. Частное в одном слове допускает максимальное значение +32 767 для беззнакового деления и +16 383 — для знакового.
- Деление учетверенного слова на двойное слово. Делимое находится в регистровой паре EDX:EAX, а делитель — в двойном слове памяти или в регистре. После деления остаток помещается в регистр EDX, а частное — в регистр EAX.

Команды div и idiv имеют единственный операнд, являющийся делителем. Рассмотрим следующую команду:

```
div DIVISOR
```

Если переменная DIVISOR определена как байт, то предполагается деление слова на байт. Если переменная DIVISDR определена как слово (DW), то операция предполагает деление двойного слова на слово. При делении, например, 13 на 3 получается результат 4 $\frac{1}{3}$. Частное будет равным 4, а остаток — 1.

Флаги состояния CF, OF, SF и ZF после выполнения команд div и idiv не определены.

В качестве примера рассмотрим деление 64-разрядного беззнакового числа на однобайтовое целое. Программный код процедуры (назовем ее _div_dd_byte) приведен в листинге 8.15.

**Листинг 8.15.** Деление беззнакового 64-разрядного числа на байт (32-разрядная версия)

```
.686
.model flat
option casemap: none
.data
  quotient  DD 0
```

```
  remainder DD 0
  dividend  DQ 1398
  divisor   DB 67
.code
_div_dd_byte proc
  movzx BX, divisor
  movzx EBX, BX
  mov   EAX, dword ptr dividend
  mov   EDX, dword ptr dividend+4
  div   EBX
  mov   quotient, EAX
  mov   remainder, EDX
  ret
 _div_dd_byte endp
 end
```

Проанализировать программу несложно. Делимое находится в 8-байтовой переменной dividend, а делитель — в переменной divisor. Вначале преобразуем делитель в двойное слово с помощью следующих двух команд:

```
movzx BX, divisor
movzx EBX, BX
```

Затем подготавливаем операцию деления. Для этого помещаем младшее двойное слово делимого в регистр EAX, а старшее двойное слово переменной dividend — в регистр EDX. Наконец, после выполнения операции деления сохраняем частное в переменной quotient, а остаток — в переменной remainder.

Деление знаковых чисел выполняется с помощью команды idiv. Модифицируем предыдущий пример таким образом, чтобы программа могла работать со знаковыми данными. Для этого нужно заменить команду div на idiv и использовать вместо команд movzx команды movsx (поскольку мы имеем дело со знаковыми операндами). Исходный текст процедуры (назовем ее _idiv_dd_byte) показан в листинге 8.16.

**Листинг 8.16.** Деление 64-разрядного знакового числа на байт (32-разрядная версия)

```
.686
.model flat
option casemap: none
.data
  _quotient  DD 0
  _remainder DD 0
  dividend   DQ 1398
  divisor    DB -93
.code
 _idiv_dd_byte proc
  movsx BX, divisor
  movsx EBX, BX
  mov   EAX, dword ptr dividend
  mov   EDX, dword ptr dividend+4
  idiv  EBX
  mov   _quotient, EAX
  mov   _remainder, EDX
  ret
 _idiv_dd_byte endp
 end
```

В результате деления получаем частное, равное –15, и остаток, равный 3.

Хочу заметить, что если делимое и делитель имеют одинаковый знаковый бит, то команды div и idiv генерируют одинаковый результат. Но если делимое и делитель имеют разные знаковые биты, то команда div генерирует положительное частное, а команда idiv — отрицательное частное.

В некоторых случаях можно добиться повышения производительности при выполнении операций деления на степень числа 2 (2, 4, и т. д.). В этом случае более эффективным является сдвиг вправо на требуемое число битов.

Рассмотрим примеры (предполагаем, что делимое находится в регистре AX) деления. Деление на 2:

```
shr  AX,1
```

Деление на 8:

```
mov  CL,3
shr  AX,CL
```

При использовании команд div и idiv может возникнуть переполнение, что вызывает прерывание. Подобная ситуация может случиться при делении на ноль, а также не исключается при делении на 1. Чтобы избежать подобных ситуаций, рекомендуется следовать таким правилам:

- если делитель — байт, то его значение должно быть меньше, чем старший байт (AH) делителя;
- если делитель — слово, то его значение должно быть меньше, чем старшее слово (DX) делителя.

Для указанных случаев частное может превысить предельно допустимое значение. Во избежание подобных ситуаций полезно выполнять проверку делителя до выполнения команд div и idiv. В следующем примере предполагается, что переменная DIVISOR является однобайтовым числом, а делимое находится в регистре AX:

```
. . .
.data
 DIVISOR DB ?
.code
 . . .
 cmp   AH, DIVISOR
 jb   overflow
 div   DIVISOR
 . . .
overflow:
  < обработчик переполнения>
 . . .
```

Во втором примере предполагаем, что DIVISOR — двухбайтовый делитель, а делимое находится в регистровой паре DX:AX. Программный код для проверки мог бы выглядеть так:

```
. . .
.data
 DIVISOR DW ?
.code
```

```
  . . .
  cmp  DX, DIVISOR
  jb  overflow
  div  DIVISOR
  . . .
overflow:
  < обработчик переполнения>
  . . .
```

Для команды idiv необходимо учитывать тот факт, что либо делимое, либо делитель может быть отрицательным, а так как сравниваются абсолютные значения, то нужно использовать команду neg для временного преобразования отрицательного значения в положительное.

Команда neg обеспечивает преобразование знака двоичных чисел из плюса в минус и наоборот. Эта особенность может быть использована при нахождении абсолютной величины (модуля) числа. В практическом плане команда neg устанавливает противоположные значения битов и прибавляет 1. Вот некоторые примеры:

```
neg AX
neg BL
neg MEMOVALUE
```

Здесь MEMOVALUE — байт или слово в памяти.

Преобразование знака для 32-разрядного (или большего) числа требует дополнительных шагов. В качестве примера рассмотрим преобразование знака для 32-разрядного числа, находящегося в регистрах DX:AX. Поскольку команда neg не может обрабатывать два регистра одновременно, то ее непосредственное применение приведет к неправильному результату. Для правильного преобразования необходимо выполнить такую последовательность команд:

```
not DX    ; инвертирование битов в DX
not AX    ; инвертирование битов в AX
add AX,1  ; прибавление 1 к AX
adc DX,0  ; прибавление переноса к DX
```

## 8.5. Обработка данных в форматах ASCII и BCD

Для получения высокой производительности компьютер выполняет арифметические операции над числами в двоичном формате. Во многих случаях новые данные вводятся программой с клавиатуры в виде ASCII-символов в десятичном формате. Аналогично, вывод информации на экран осуществляется в ASCII-кодах. Например, число 23 в двоичном представлении выглядит как 00010111, а в шестнадцатеричном — как 17h. В ASCII-коде на каждый символ требуется один байт, поэтому число 25, например, в ASCII-коде имеет внутреннее представление 3235h.

Далее мы рассмотрим технику преобразования данных из формата ASCII в двоичный формат для выполнения арифметических операций и обратного преобразования двоичных результатов в формат ASCII для вывода на экран или принтер.

Как уже было сказано, данные, вводимые с клавиатуры, имеют формат ASCII, например, буквы INT имеют в памяти шестнадцатеричное представление 494E54,

цифры 1234 — шестнадцатеричное представление 31 323 334. В большинстве случаев формат символьных данных, например фрагментов текста, программой не изменяется. В то же время для выполнения арифметических операций над числовыми величинами необходима специальная обработка.

Процессор позволяет производить такую обработку с помощью специальных ассемблерных команд, предназначенных для выполнения арифметических операций непосредственно над числами в формате ASCII:

- aaa (ASCII Adjust for Addition) — коррекция для сложения ASCII-кода;
- aad (ASCII Adjust for Division) — коррекция для деления ASCII-кода;
- aam (ASCII Adjust for Multiplication) — коррекция для умножения ASCII-кода;
- aas (ASCII Adjust for Subtraction) — коррекция для вычитания ASCII-кода.

Эти команды кодируются без операндов и выполняют автоматическую коррекцию содержимого регистра AX. Коррекция необходима, так как ASCII-код представляет собой так называемый неупакованный десятичный формат, в то время как компьютер выполняет арифметические операции в двоичном формате.

Рассмотрим более детально процесс сложения чисел в формате ASCII, а именно пример сложения чисел 8 и 4. В шестнадцатеричном формате эти числа равны 38h и 34h. Их сумма равна 6Ch, что является неправильным значением ни для формата ASCII, ни для двоичного формата.

Если не брать во внимание левый полубайт (6) и прибавить 6 к правому полубайту (0Ch), то получим 12h, что является правильным результатом, если рассматривать это значение в десятичном формате. Это означает, что путем определенных преобразований можно получить правильный двоично-десятичный формат результата сложения. Рассмотрим практический пример, иллюстрирующий вышесказанное. Программный код примера выглядит так:

```
. . .
.data
  num1  DB 34h
  num2  DB 38h
.code
  . . .
  xor  AX, AX
  mov  AL, num1
  mov  BL, num2
  add  AL, BL
  aaa
 . . .
```

Из примера видно, что регистр AL содержит значение 38h, а регистр BL — 34h. Числа 38h и 34h представляют собой два байта в формате ASCII. После выполнения сложения (команда add AL, BL) регистр AL будет содержать значение 6Ch. После выполнения коррекции (команда aaa) регистр AX будет содержать неупакованное двоично-десятичное число 0102h.

Команда aaa проверяет правый полубайт (4 бита) в регистре AL. Если его значение находится между A и F или флаг AF равен 1, то к регистру AL прибавляется 6,

к регистру AH прибавляется 1, а флаги AF и CF устанавливаются в 1. Кроме того, команда aaa устанавливает в 0 левый полубайт в регистре AL.

Для того чтобы получить символьное представление ASCII-числа, необходимо выполнить операцию поразрядного ИЛИ над левым полубайтом результата с помощью команды or. Для регистра AX эта операция будет выглядеть так:

```
or AX. 3030H
```

Рассмотренный пример демонстрирует основные принципы сложения однобайтовых чисел в формате ASCII. Сложение многобайтовых ASCII-чисел требует организации цикла, который выполняет обработку справа налево с учетом переноса.

Следующие примеры представляют собой небольшие программы, демонстрирующие различные аспекты сложения ASCII-чисел, а также вывод результатов на экран дисплея или их передачу другим программам. Первый пример иллюстрирует сложение двух однобайтовых ASCII-чисел и вывод результата на экран. Это простое 16-разрядное приложение MS-DOS, программный код которого показан в листинге 8.17.

**Листинг 8.17.** Сложение двух однобайтовых чисел в ASCII-представлении (16-разрядная версия)

```
.model small
.data
  num1 DB '9'
  num2 DB '5'
  sum  DB 2 DUP (' ')
.code
start:
  mov  AX. @data
  mov  DS. AX
  clc                    : очистка флага переноса и регистра AX
  xor  AX. AX
  mov  AL. num1          : заносим первое число в AL
  adc  AL. num2          : сложение с num2 с учетом переноса
  aaa                    : коррекция результата
  or   AX. 3030h         : преобразование в символьное представление
  xchg AH. AL            : обмен байтами для подготовки вывода на экран
  mov  word ptr sum. AX  : сохранение результата в переменной sum
                         : вывод результата на экран
  mov CX. 2              : помещаем число выводимых байтов в регистр CX
  mov AH. 6h
  lea SI. sum            : помещаем адрес переменной sum в SI
 next:
  mov DL. byte ptr [SI]  : помещаем выводимый байт в регистр DL
  int 21h                : вывод на экран
  inc SI                 : заносим адрес следующего символа в SI
  loop next              : повтор цикла
  mov ax. 4c00h
  int 21h
end start
end
```

Пример сложения ASCII-чисел в 32-разрядных Windows-приложениях выглядит сложнее. Здесь выполняется сложение четырехбайтовых чисел num1 и num2, а результат помещается в переменную sum. Операция сложения реализована в виде процедуры _add_asc, возвращающей адрес результата в 32-разрядном регистре EAX. Операция сложения выполняется, начиная с младших байтов (они расположены в старших адресах переменных num1 и num2). После выполнения процедуры переменная sum содержит значение 1023.

Процедуру можно усовершенствовать и использовать как в программах на ассемблере, так и в приложениях, написанных на языках высокого уровня. Исходный текст процедуры _add_asc показан в листинге 8.18.

**Листинг 8.18.** Сложение ASCII-чисел (32-разрядная версия)

```
.686
.model flat
option casemap:none
.data
  num1  DB '0037'
  len1  EQU $-num1
  num2  DB '0986'
  sum   DB 4 DUP (' ')      ; резервируем память для результата
.code
 _add_asc proc
   mov  ECX, len1           ; помещаем размер операндов (в байтах) в ECX
   clc                      ; очистка флага переноса
                            ; побайтовое сложение в цикле
 again:
   mov  AL, byte ptr num1[ECX-1] ; помещаем младший байт num1 в AL
   adc  AL, byte ptr num2[ECX-1] ; сложение с таким же байтом в num2
   aaa                           ; коррекция результата
   mov  byte ptr sum[ECX-1], AL  ; сохранение результата
                                 ; в соответствующем байте переменной
                                 ; sum
   loop again
   adc  byte ptr sum[ECX-1], 0   ; коррекция результата
   or   dword ptr sum, 30303030h ; преобразование в символьный вид
   lea  EAX, sum                 ; сохраним адрес результата в регистре
                                 ; EAX
   ret
 _add_asc endp
 end
```

Перейдем к более детальному рассмотрению операций вычитания чисел, представленных в формате ASCII. При вычитании таких чисел, так же как и при сложении, требуется коррекция результата. Именно этим целям и служит команда aas.

Она выполняется аналогично команде aaa. Команда aas проверяет правый полубайт в регистре AL. Если его значение находится в диапазоне A – F или флаг AF равен 1, то из регистра AL вычитается 6, а из регистра AH вычитается 1, при этом флаги AF и CF устанавливаются в 1. Во всех случаях команда aas помещает нули в левый полубайт регистра AL.

Следующие примеры демонстрируют использование команды aas. Исходный текст первого примера выглядит так:

```
 . . .
.data
  op1   DB 38h
  op2   DB 34h
.code
  . . .
  mov  AL, op1 ; 38h в AL
  sub  AL, op2 ; AL - 38h = 04h
  aas          ; AL после коррекции: 04h
  . . .
```

В этом примере коррекция команде aas не требуется. Следующий пример демонстрирует более сложный случай:

```
 . . .
.data
  op1   DB 38h
  op2   DB 34h
.code
  . . .
  mov  AL, op2 ; 34h в AL
  sub  AL, op1 ; AL - 34h = 0FCh
  aas          ; AX после коррекции: 0FF06h
  . . .
```

В этом примере из-за того, что правая цифра в регистре AL равна 0Ch, команда aas вычитает 6 из регистра AL и 1 из регистра AH и устанавливает в 1 флаги AF и CF. Результат равен –4 (0FF06h), то есть десятичное дополнение числа 4.

Далее рассмотрим более сложные примеры вычитания чисел в формате ASCII. Вычитание однобайтовых ASCII-чисел продемонстрировано в приложении MS-DOS, программный код которого напоминает рассмотренный ранее пример 16-разрядной программы сложения (см. листинг 8.17). Модифицированная версия имеет следующие отличия: команда сложения adc здесь заменена командой вычитания sbb, а для коррекции результата вместо aaa применяется команда aas. Исходный текст примера показан в листинге 8.19.

**Листинг 8.19.** Вычитание ASCII-чисел (16-разрядная версия)

```
.model small
.data
  num1    DB '8'
  num2    DB '3'
.code
start:
  mov  AX, @data
  mov  DS, AX
  clc              ; очистка флага переноса
  mov  AL, num1    ; первое число в AL
  sbb  AL, num2    ; вычесть второе с учетом возможного заема
  aas              ; коррекция результата
  or   AL, 30h     ; преобразовать результат в символьное представление
                   ; вывод на экран
  mov  DL, AL
```

продолжение

**Листинг 8.19** (продолжение)

```
  mov AH, 6h
  int 21h
  mov ax, 4c00h
  int 21h
end start
end
```

Следующая 32-разрядная процедура также демонстрирует вычитание ASCII-чисел. Процедура называется _sub_asc, а программный код напоминает тот, что используется в процедуре _add_asc (см. листинг 8.18). Различие состоит в том, что вместо соответствующих команд для сложения применяются команды для вычитания. Кроме того, операнд num1 должен быть больше num2. При желании процедуру _sub_asc можно модифицировать для работы с произвольными операндами. Исходный текст процедуры представлен в листинге 8.20.

**Листинг 8.20.** Вычитание ASCII-чисел (32-разрядная версия)

```
.686
.model flat
option casemap:none
.data
  num1  DB '0737'
  len1  EQU $-num1
  num2  DB '0086'
  subs  DB 4 DUP (' ')
.code
 _sub_asc proc
   mov  ECX, len1
   clc
 again:
   mov  AL, byte ptr num1[ECX-1]
   sbb  AL, byte ptr num2[ECX-1]
   aas
   mov  byte ptr subs[ECX-1], AL
   loop again
   sbb  byte ptr subs[ECX-1], 0
   or   dword ptr subs, 30303030h
   lea  EAX, subs
   ret
 _sub_asc endp
 end
```

Думаю, что нет смысла комментировать этот пример, поскольку он похож на ранее рассмотренный пример сложения чисел.

Следующая тема, которую мы рассмотрим, — умножение чисел, представленных в формате ASCII. При умножении ASCII-чисел результат уже не будет являться ASCII-числом. Для преобразования результата в формат ASCII необходимо выполнить дополнительно команду aam.

Команда aam выполняется после умножения двух неупакованных двоично-десятичных чисел. Она преобразует результат умножения, являющийся двоичным числом, в правильное неупакованное двоично-десятичное число, младший разряд которого помещается в регистр AL, а старший — в AH.

Следует заметить, что коррекция осуществляется только для одного байта за одну операцию, поэтому можно умножать только однобайтовые поля — для более длинных полей необходима организация цикла.

Команда aam делит содержимое регистра AL на 10 и помещает частное в регистр AH, а остаток — в AL. Рассмотрим простой пример:

```
. . .
.data
  op1  DB 37
  op2  DB 39
.code
. . .
mov  AL, op1
mov  CL, op2
and  CL,0FH       ; преобразовать содержимое CL в неупакованный формат
                  ; (CL = 09)
and  AL,0FH       ; преобразовать содержимое AL в неупакованный формат
                  ; (AL = 07)
mul  CL           ; умножить AL на CL
aam               ; преобразовать содержимое AX в неупакованный
                  ; двоично-десятичный формат
or      AX,3030H  ; преобразовать содержимое AX в формат ASCII
. . .
```

После выполнения команды mul в регистре AX будет находиться число 63 (003Fh). Далее команда aam делит это число на 10, записывая частное 06 в регистр AH и остаток 03 в регистр AL. После этого команда or выполняет преобразование неупакованного десятичного числа в формат ASCII.

Рассмотрим деление чисел, представленных в формате ASCII. При делении ASCII-чисел применяется команда aad, которая выполняет корректировку ASCII-кода делимого непосредственно перед выполнением операции деления с помощью команды div. Перед использованием команды aad необходимо обнулить левые полубайты ASCII-кодов для получения неупакованного десятичного формата. Следует отметить, что команда aad может оперировать двухбайтовым делимым в регистре AX.

Следующий фрагмент программного кода наряду с подготовкой операндов к делению выполняет коррекцию для последующего деления. Предположим, что переменная op1 содержит делимое 3238h в формате ASCII, а переменная op2 — делитель 37h также в формате ASCII. Программный код, выполняющий деление ASCII-чисел, выглядит так:

```
. . .
.data
 op1   DW 3238h
 op2   DB  37h
.code
  . . .
  xor   AX, AX
  mov   AX, op1
  mov   CL, op2
  and   AX, 0F0Fh ; преобразовать AX в неупакованное число
  and   CL, 0Fh   ; преобразовать CL в неупакованное число
  aad             ; преобразовать AX в двоичный формат
  div   CL        ; деление на 7
  . . .
```

Команда aad умножает содержимое AH на 10, после чего прибавляет результат 20 к регистру AL и очищает регистр AH.

Следующий пример представляет собой законченное 16-разрядное приложение MS-DOS, демонстрирующее операцию деления ASCII-чисел и вывод результата на экран (листинг 8.21).

**Листинг 8.21.** Деление ASCII-чисел (16-разрядная версия)

```
.model small
.data
  num1    DB '18'
  num2    DB '2'
  res     DB  ?
.code
start:
  mov  AX, @data
  mov  DS, AX
  clc
  mov  AX, word ptr num1   ; поместить делимое в AX
  xchg AH, AL              ; установить правильный порядок байтов
  and  AX, 0F0Fh           ; преобразовать число в AX в неупакованное
                           ; двоично-десятичное (AX = 0306h)
  mov  CL, num2            ; поместить второе число в CL
  and  CL, 0Fh             ; преобразовать число в CX в неупакованное
                           ; двоично-десятичное (CL = 02h)
  aad                      ; выполнить коррекцию AX перед делением
                           ; (AX = 1Ch)
  div  CL
  or   AL, 30h             ; преобразовать результат в символьное
                           ; ASCII-представление (AL = 34h)
  mov  DL, AL              ; вывод символа на экран
                           ; (в DL - выводимый символ)
  mov AH, 2h
  int 21h
  mov ax, 4c00h
  int 21h
end start
end
```

Следующая тема, которую мы рассмотрим, — операции с упакованными двоично-десятичными числами. Как уже упоминалось ранее, упакованные числа можно только складывать и вычитать без каких-либо промежуточных преобразований. Но результат операций нужно корректировать с помощью специальных команд:

- daa (Decimal Adjustment for Addition) — десятичная коррекция для сложения;
- das (Decimal Adjustment for Subtraction) — десятичная коррекция для вычитания.

Команда daa преобразует двоичный результат выполнения команд add и adc в регистре AL в упакованное десятичное число. Продемонстрируем работу этой команды на примере:

```
 . . .
.data
  op1  DB 32h
```

```
 op2  DB 59h
.code
 . . .
 mov AL. opl
 add AL. op2  ; AL = 8Bh
 daa          ; AL = 91h
 . . .
```

После сложения операндов opl и op2 в регистре AL будет число 8Bh. Младшая цифра результата больше 9, поэтому она преобразуется. Конечный результат равен 91h, что и требовалось получить.

Команда das преобразует двоичный результат выполнения команд sub и sbb в регистре AL в упакованное десятичное число. Следующий пример демонстрирует применение этой команды:

```
 . . .
.data
 opl  DB 37h
 op2  DB 51h
.code
 . . .
 moy AL. op2
 mov CL. opl
 sub AL.  CL  ; AL = 1Ah
 das          ; AL = 14h
 . . .
```

# 8.6. Преобразование ASCII-чисел в двоичный формат

Выполнение арифметических операций над числами в формате ASCII или BCD удобно лишь для коротких полей. В большинстве случаев для арифметических операций требуется преобразование в двоичный формат. На практике намного проще преобразовать ASCII-число непосредственно в двоичный формат, чем переводить формат ASCII сначала в формат BCD, а затем в двоичный формат.

Метод преобразования базируется на том, что формат ASCII имеет основание 10, а компьютер выполняет арифметические операции только над числами с основанием 2. Алгоритм преобразования состоит в том, что, начиная с самого правого байта ASCII-числа, выполняется такая последовательность шагов:

1. Устанавливается в 0 левый полубайт каждого байта ASCII-числа.
2. ASCII-цифры умножаются на 1, 10, 100, и результаты складываются.

Для примера рассмотрим преобразование числа 2459 из формата ASCII в двоичный формат. Полагаем, что результат преобразования будет храниться в регистре AX. Вначале обнуляем регистр AX, после чего считываем цифру в ASCII-кодировке, умножаем AX на 10 и прибавляем двоичное значение цифры в AX. После считывания всех цифр ASCII-числа 2459 в регистре AX будет содержаться двоичное значение числа 2459. Для наглядности можно представить процесс преобразования в виде таблицы (табл. 8.1).

**Таблица 8.1.** Преобразование ASCII-числа в двоичное число

| Регистр AX перед итерацией | | Вновь извлекаемое число | | AX после итерации |
|---|---|---|---|---|
| 0 × 10 | + | 2 | = | 2 |
| 2 × 10 | + | 4 | = | 24 |
| 24 × 10 | + | 5 | = | 245 |
| 245 × 10 | + | 9 | = | 2459 |

Для лучшего понимания алгоритма преобразования приведу пример процедуры (назовем ее _asc_bin), в которой определяется сумма двух чисел: одно представлено в формате ASCII, другое является обычным двоичным числом. Сумма возвращается в регистре EAX. Подобную процедуру можно использовать в 32-разрядных приложениях, разработанных на ассемблере или на языках высокого уровня. Исходный текст программного кода процедуры _asc_bin показан в листинге 8.22.

**Листинг 8.22.** Вычисление суммы ASCII-числа и двоичного числа (32-разрядная версия)

```
.686
.model flat
option casemap: none
.data
 i_asc   DB  '5749'
 len     EQU $-i_asc         ; определяем размер ASCII-числа в байтах
 i_bin   DD 3772
.code
 _asc_bin proc
   clc                       ; очистка флага переноса
   lea   ESI, i_asc          ; помещаем адрес i_asc в ESI
   mov   ECX, len            ; сохраняем размер числа в ECX
   xor   EAX, EAX
   mov   BX,  10             ; помещаем множитель в регистр BX
 again:
   mul   BX                  ; AX * BX
   mov   DL, byte ptr [ESI]  ; загружаем очередную цифру ASCII-числа в DL
   and   DL, 0Fh             ; очищаем левый полубайт
   movzx DX, DL              ; расширяем DL до DX для последующего
                             ; сложения
   add   AX, DX              ; сложение частичной суммы и преобразованной
                             ; ASCII-цифры
   inc   ESI                 ; переход к следующему байту ASCII-числа
   loop  again               ; следующая итерация
   movzx EAX, AX             ; расширение AX до EAX для выполнения
                             ; 32-разрядного сложения
   add   EAX, i_bin          ; вычисление суммы i_asc + i_bin
                             ; и сохранение ее EAX
   ret
 _asc_bin endp
  end
```

При данных значениях переменных i_asc и i_bin результат сложения будет равен 9521. Процедуру можно вызвать из программы на Visual C++ .NET (листинг 8.23).

**Листинг 8.23.** Демонстрационная программа для процедуры из листинга 8.22

```
#include <stdio.h>
extern "C" int asc_bin(void):
int main(void)
{
 printf("ASCII-number + binary = %d\n". asc_bin()):
 return 0:
}
```

# 8.7. Преобразование двоичных чисел в формат ASCII

Для того чтобы напечатать или отобразить на экране арифметический результат, необходимо преобразовать его в формат ASCII. Данная операция включает в себя процесс, обратный предыдущему: вместо умножения выполняется деление двоичного числа на 10 до тех пор, пока результат не станет меньше 10. Остатки, находящиеся в диапазоне 0–9, образуют число в формате ASCII.

В качестве примера рассмотрим демонстрационную процедуру (назовем ее _bin_asc_5), в которой выполняется преобразование двоичного числа в его ASCII-представление, после чего полученное ASCII-число суммируется с другим ASCII-числом, равным 5. Процедура возвращает в регистре EAX 32-разрядный адрес символьной строки, содержащей результат сложения ASCII-чисел. Исходный текст процедуры представлен в листинге 8.24.

**Листинг 8.24.** Преобразование числа в формат ASCII с последующим сложением (32-разрядная версия)

```
.686
.model flat
option casemap: none
.data
  op_bin  dd 273
  op_asc  db  3 dup (' ')
.code
 _bin_asc_5 proc
   lea  ESI. op_asc+2
   mov  EAX. op_bin
   mov  EBX. 10
 next:
   xor  EDX. EDX
   div  EBX
   or   EDX. 30h
   mov  byte ptr [ESI]. DL
   cmp  EAX. 10
   jl   complete
   dec  ESI
   jmp  next
 complete:
   or   EAX. 30h
   dec  SI
```

продолжение

**Листинг 8.24** (продолжение)

```
    mov  byte ptr [ESI]. AL
    clc
    mov  AL. byte ptr op_asc+2
    adc  AL. '5'
    aaa
    or   AL. 30h
    mov  byte ptr op_asc+2. AL
    lea  EAX. op_asc
    ret
 _bin_asc_5 endp
  end
```

# 8.8. Полезные алгоритмы и программы

В этом разделе приведены примеры вычислительных процедур, с которыми программисту приходится часто сталкиваться на практике. При решении повседневных задач в большинстве случаев нужно иметь дело не с одиночными числами, а с их множествами, собранными в массивы. Очень часто требуется вычислять сумму и разность элементов целочисленных массивов, осуществлять поиск минимального и максимального значений в массиве, проверять вычисление модулей элементов массива и т. д. Приведенные далее примеры могут помочь в решении таких задач. Разработчик легко может их модифицировать для своих потребностей.

Первый пример, который мы рассмотрим, — вычисление суммы элементов двух целочисленных массивов. Предположим, имеется два целочисленных массива, a1 и a2, состоящие из четырех элементов. Результаты попарного сложения элементов массивов заносятся в третий массив sum. Программный код, выполняющий операции сложения, реализован в процедуре _sum_ints (листинг 8.25).

**Листинг 8.25.** Сложение элементов целочисленных массивов (32-разрядная версия)

```
.586
.model flat
option casemap: none
.data
  a1  DD 12. -345. -49. 91
  a2  DD -48. 199. -812. 32
  len EQU $-a2
  sum DD 4 DUP (0)
.code
 _sum_ints proc
   mov ECX. len      : размер массивов (в байтах) -> ECX
   shr ECX. 2        : коррекция ECX для операций с двойными словами
                     :(деление на 4)
   lea ESI. a1       : адрес первого элемента массива a1 -> ESI
   lea EDI. a2       : адрес первого элемента массива a2 -> EDI
   lea EBX. sum      : адрес первого элемента массива sum -> EBX
 next:
   clc
   xor EAX. EAX      : перед выполнением операций очищаем регистр EAX
   mov EAX. [ESI]    : элемент массива a1 -> EAX
   adc EAX. [EDI]    : сложить с соответствующим элементом массива a2
```

```
  mov [EBX], EAX    ; помещаем сумму элементов на соответствующую позицию
                    ; в массиве sum
  add ESI, 4        ; переход к адресу следующего элемента в массиве a1
  add EDI, 4        ; переход к адресу следующего элемента в массиве a2
  add EBX, 4        ; переход к адресу следующего элемента в массиве sum
  loop next         ; следующая итерация
  lea  EAX, sum     ; адрес массива sum -> EAX
  ret
_sum_ints endp
 end
```

Думаю, исходный текст процедуры не вызывает затруднений. Хочу лишь обратить внимание, что поскольку здесь выполняются операции над двойными словами, то счетчик ECX должен содержать их количество (в данном случае — 4). Кроме того, следует учитывать, что последующие элементы массивов размещены по адресам, на 4 большим текущего.

Если нужно складывать элементы, представленные словами, то некоторые команды в данной процедуре следует изменить:

```
shr ECX, 2
```

Вместо этой команды нужно использовать такую:

```
shr ECX, 1
```

Кроме того, при переходе к следующим элементам массивов требуются команды

```
add ESI, 2
add EDI, 2
add EBX, 2
```

Процедуру можно легко модифицировать для работы с произвольным количеством элементов в массивах.

Для большинства практических применений размерности двойного числа достаточно для получения корректного результата. Однако иногда приходится иметь дело с очень большими числами, для которых разрядности двойного слова может оказаться мало. Следующий пример демонстрирует вычисление сумм 64-разрядных элементов целочисленных массивов a1 и a2. Результат, как и в предыдущем примере, помещается в массив sum. Операция сложения реализована в процедуре _sum_ints_64, которая возвращает в регистре EAX адрес массива sum (листинг 8.26).

**Листинг 8.26.** Сложение 64-разрядных элементов массивов (32-разрядная версия)

```
.586
.model flat
option casemap: none
.data
  a1  DQ 123980127, -1296432345, -971743249, 9740391
  a2  DQ -48094715, 81199054, -81283467, 340917622
  len EQU $-a2
  sum DQ  4 DUP (0)
.code
 _sum_ints_64 proc
   mov ECX, len
   shr ECX, 3
   lea ESI, a1
   lea EDI, a2
   lea EBX, sum
```

продолжение

**Листинг 8.26** (продолжение)

```
next:
  clc
  xor EAX, EAX
  mov EAX, dword ptr [ESI]
  add EAX, dword ptr [EDI]
  mov dword ptr [EBX], EAX
  mov EAX, dword ptr [ESI+4]
  adc EAX, dword ptr [EDI+4]
  mov dword ptr [EBX+4], EAX
  add ESI, 8
  add EDI, 8
  add EBX, 8
  loop next
  lea  EAX, sum
  ret
_sum_ints_64 endp
 end
```

Особенностью работы этой процедуры является то, что сложение 64-разрядных операндов реализовано черэ попарное сложение 32-разрядных частей этих операндов. Алгоритм сложения построен следующим образом: вначале складываются младшие 32-разрядные части, затем — старшие с учетом возможного переноса из младшей части. После этого младшая и старшая части 64-разрядного результата сохраняются в младшей и старшей частях соответствующего элемента массива sum.

Переход к следующим элементам массивов выполняется путем прибавления значения 8 к указателям на текущие элементы. В счетчике ECX содержится количество 8-байтовых элементов. При указанных значениях элементов массивов a1 и a2 элементы в массиве sum равны 75 885 412, –1 215 233 291, –1 053 026 716, 350 658 013. Такую процедуру можно модифицировать для работы с произвольным количеством элементов массивов.

Процедура может быть использована в вычислительных алгоритмах для больших чисел в различных программах. Например, следующее консольное 32-разрядное приложение на Visual C++ .NET выполняет вывод на экран результатов работы процедуры _sum_ints_64 (листинг 8.27).

**Листинг 8.27.** Демонстрационная программа для процедуры из листинга 8.26

```
#include <stdio.h>
extern "C" long long int* sum_ints_64(void);
int main(void)
{
  long long int* plong = sum_ints_64();
  printf("\n\n\tSum of long array:\n");
  for (int i1 = 0; i1 < 4; i1++)
   {
    printf("%lld ", *plong++);
   };
  return 0;
}
```

Рассмотрим еще одну задачу, с которой часто сталкиваются программисты, — вычисление суммы элементов целочисленного массива. Реализация алгоритма нахождения суммы 64-разрядных элементов показана в процедуре _sum_64 (листинг 8.28).

**Листинг 8.28.** Вычисление суммы 64-разрядных элементов целочисленного массива (32-разрядная версия)

```
.586
.model flat
option casemap: none
.data
  iarray  DQ  9234764129, -16097233481, -7565902112, 39094647921
  len     EQU $-iarray
  sum     DQ  0
.code
 _sum_64 proc
   mov ECX, len     ; размер массива iarray (в байтах) -> ECX
   shr ECX, 3       ; преобразовать размер в количество учетверенных слов
   lea ESI, iarray  ; адрес массива -> ESI
   lea EDI, sum     ; адрес переменной sum -> EDI
   clc
 next:
   mov EAX, dword ptr [ESI]     ; младшее двойное слово элемента
                                ; массива-> EAX
   add dword ptr [EDI],EAX      ; прибавить к младшему двойному слову
                                ; общей суммы
   mov EDX, dword ptr [ESI+4]   ; старшее двойное слово элемента
                                ; массива -> EDX
   adc dword ptr [EDI+4], EDX   ; прибавить к старшему двойному слову
                                ; общей суммы с учетом переноса
   add ESI, 8                   ; переход к следующему элементу массива
   clc                          ; очистить флаг переноса
   loop next                    ; следующая итерация
   lea  EAX, sum                ; адрес переменной sum -> EAX
   ret
 _sum_64 endp
 end
```

В основе алгоритма суммирования элементов массива лежит тот же подход, что и при суммировании 64-разрядных элементов двух массивов в предыдущем примере. Текущее значение суммы помещается в переменную sum. В каждой итерации к сумме добавляется значение следующего элемента массива. Вначале прибавляется младшее двойное слово, затем с учетом переноса — старшее двойное слово. По завершении цикла адрес переменной sum помещается в регистр EAX и возвращается в основную программу.

При указанных значениях элементов массива после выполнения операции суммирования в переменной sum будет находиться значение 24 666 276 457.

Еще одной весьма распространенной задачей является поиск наибольшего или наименьшего элемента в массиве чисел. С помощью программы на ассемблере можно довольно просто реализовать алгоритм такого поиска. Следующая процедура (назовем ее _max) позволяет найти максимальный элемент в массиве целых чисел. Процедура возвращает адрес переменной, содержащей максимальный по значению элемент, в регистре EAX. Исходный текст программы показан в листинге 8.29.

**Листинг 8.29.** Поиск наибольшего элемента в массиве целых чисел (32-разрядная версия)

```
.586
.model flat
option casemap: none
.data
```

продолжение ➩

**Листинг 8.29** (продолжение)

```
 iarray DD 45, -78, 34, 9, 231, 45, -12
 len    EQU $-iarray
 maxval DD 0
.code
 _max proc
  lea  ESI, iarray  ; адрес первого элемента -> ESI
  mov  ECX, len     ; размер массива -> ECX
  shr  ECX, 2       ; скорректировать размер
  mov  EAX, [ESI]   ; поместить первый элемент массива в EAX
                    ; и принять его в качестве максимума
next:
  cmp  EAX, [ESI+4] ; сравнить со следующим элементом массива
  jl   change       ; если EAX меньше, заменить его
go_loop:
  add  ESI, 4       ; перейти к следующему элементу
  loop next
  jmp  exit
change:
  mov  EAX, [ESI+4]
  jmp  go_loop
exit:
  mov  max, EAX
  lea  EAX, maxval
  ret
 _max  endp
  end
```

Алгоритм поиска реализован следующим образом:

1. В регистр EAX помещается первый элемент массива iarray, который считается первым максимумом.
2. Содержимое регистра EAX сравнивается со следующим элементом массива:
   * если значение в EAX больше значения элемента массива, то выбирается следующий элемент для сравнения и цикл повторяется;
   * если значение в EAX меньше значения элемента массива, то в регистр помещается новый максимум.

После окончания цикла полученное максимальное значение помещается в переменную maxval, после чего адрес переменной передается в регистр EAX. Процедуру легко модифицировать для поиска минимального элемента массива. Вот фрагмент измененного кода (изменения выделены жирным шрифтом):

```
 . . .
next:
  cmp  EAX, [ESI+4]
  jg   change
go_loop:
 . . .
```

Последний пример, который мы рассмотрим, — вычисление сумм всех положительных и отрицательных чисел, находящихся в массиве целых чисел. Сумма положительных чисел массива a1 сохраняется в 32-разрядной переменной sum_plus, а сумма отрицательных чисел помещается в переменную sum_minus. Вычислительный алгоритм реализован в процедуре _sum_plus_minus, исходный текст которой представлен в листинге 8.30.

**Листинг 8.30.** Вычисление сумм положительных и отрицательных элементов массива целых чисел (32-разрядная версия)

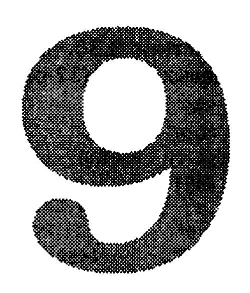
```
.586
.model flat
option casemap: none
.data
  a1        DD 123, -96, -17, 403
  len       EQU $-a1
  res       label dword
  sum_plus  DD 0
  sum_minus DD 0
.code
 _sum_plus_minus proc
   mov ECX, len    ; помещаем размер массива a1 в регистр ECX
   shr ECX, 2      ; корректируем счетчик для двойных слов
   lea EBX, a1     ; адрес массива a1 -> EBX
next:
   xor EAX, EAX
   mov EAX, dword ptr [EBX] ; очередной операнд -> EAX
   cmp EAX, 0               ; сравнить с нулем
   jl  plus                 ; если число больше 0, прибавить его
                            ; к sum_plus
   add sum_plus, EAX
   jmp continue
 plus:
   adc sum_minus, EAX       ; число меньше 0, прибавить его
                            ; к sum_minus
 continue:
   add EBX, 4               ; переход к следующему элементу массива
   loop next                ; следующая итерация
   lea  EAX, res            ; адрес результата -> EAX
   ret
 _sum_plus_minus endp
 end
```

Заканчивая обзор операций целочисленной арифметики, можно сделать некоторые выводы:

- Во всех операциях лучше всего использовать операнды большей размерности, даже если исходные значения укладываются в переменные меньшей размерности. Результат операции может превысить размерность операндов, что вызовет ошибки переполнения.
- По возможности следует выполнять операции над однотипными операндами (байт-байт, слово-слово и т. д.). Это позволит избежать необходимости преобразования типов с помощью команд cbw, cwd, cdq, movsx, movzx, которые отнимают часть процессорного времени и замедляют работу программы в целом. Если такие преобразования выполняются в цикле, состоящем из десятков тысяч итераций, то замедление может оказаться существенным.

Если нужно выполнять умножение или деление на числа, кратные 2, то лучше использовать для этой цели команды логического и арифметического сдвига. Это дает существенный выигрыш в скорости.

# Использование математического сопроцессора



До сих пор мы рассматривали выполнение арифметических операций над целочисленными значениями с помощью базовых команд процессора Intel Pentium, таких, как add, sub, mul, div и т. д. В самом процессоре эти операции выполняют модули целочисленных операций. В то же время большая часть вычислений требует использования вещественных чисел или, как принято говорить, чисел с плавающей точкой. Операции над числами с плавающей точкой можно также выполнять при помощи команд целочисленной арифметики, создавая специальные алгоритмы.

Однако намного проще и эффективнее использовать для этих целей математический сопроцессор, или просто сопроцессор. Для создания вычислительных алгоритмов обработки чисел с плавающей точкой необходимо достаточно хорошо знать систему команд и особенности работы сопроцессора или, как его еще называют, модуля операций с плавающей точкой (Floating Point Unit, FPU) процессоров Intel.

Сопроцессор обрабатывает команды с плавающей точкой, отслеживая команды процессора и работая параллельно с ним (рис. 9.1).

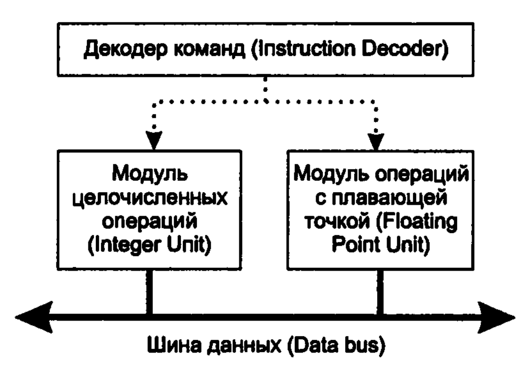



**Рис. 9.1.** Взаимодействие модуля целочисленной обработки и FPU

Параллельная работа процессора и сопроцессора повышает производительность выполнения программ в целом. Сопроцессор расширяет математические возможности основного процессора, но не замещает ни одну из его команд. Основные арифметические команды, такие, как add, sub, mul, div, и другие, выполняются процессором, а математический сопроцессор выполняет дополнительные более эффективные команды арифметической обработки. С точки зрения программиста, система с математическим сопроцессором выглядит как единый процессор с расширенным набором команд.

В этой главе основное внимание уделено практическим аспектам работы сопроцессора и использованию его возможностей при разработке программ. Полное описание сопроцессора могло бы занять отдельную книгу, поэтому рассмотрены наиболее важные моменты в функционировании этого модуля. Теоретические основы построения сопроцессора, а также принципы представления данных более подробно описаны в фирменных руководствах Intel, к которым можно обратиться для более глубокого изучения темы.

Материал главы снабжен многочисленными примерами подпрограмм на ассемблере, которые могут использоваться как в других ассемблерных программах, так и в программах на языках высокого уровня. Все подпрограммы предназначены для обработки 32-разрядных данных. Для проверки результатов работы подпрограмм на ассемблере можно использовать простейшие программы на Visual C++ .NET, хотя без каких-либо ограничений подойдут и более ранние версии компиляторов, например Visual C++ версии 6.

Я сознательно не задействую для просмотра результатов работы процедур отладчики, поскольку это отнимает много времени, а сам результат далеко не очевиден и нуждается в дополнительной интерпретации. К тому же это требует хороших навыков в отладке программ, что вызывает дополнительные трудности.

Знакомство с работой математического сопроцессора начнем с обзора типов данных, которые сопроцессор может обрабатывать.

## 9.1. Типы данных сопроцессора

Математический сопроцессор расширяет номенклатуру форматов данных, с которыми работает основной процессор. К таким форматам относятся:

- целые двоичные числа разрядности 16, 32 и 64 бит;
- упакованные целые десятичные (BCD) числа размером до 9 байт;
- вещественные числа в коротком (32 бита), длинном (64 бита) и расширенном (80 бит) форматах.

Помимо этих форматов поддерживаются и специальные числовые значения, к которым относятся:

- денормализованные вещественные числа;
- нуль;
- положительные и отрицательные значения бесконечности;
- нечисла;
- различного рода неопределенности и неподдерживаемые форматы.

Сопроцессор имеет единое внутреннее представление данных и хранит все числа в едином 80-разрядном расширенном формате. Это один из форматов представления вещественных чисел, который в точности соответствует формату регистров стека сопроцессора. Любые операнды, представленные в виде 16-, 32- и 64-разрядных целых чисел, 32-, 64- или 80-разрядных чисел с плавающей точкой, а также упакованных BCD-чисел, представленных 18 цифрами, при загрузке в регистры сопроцессора автоматически переводятся в расширенный формат. Результаты вычислений переводятся обратно в один из этих форматов данных и сохраняются в регистрах или памяти.

Рассмотрим более детально типы данных сопроцессора и начнем с двоичных чисел. Они могут иметь один из трех форматов:

- целое число размером 16 бит, диапазон значений от –32768 до +32767;
- короткое целое число размером 32 бит, диапазон значений от $-2 \times 10^9$ до $+2 \times 10^9$;
- длинное целое число размером 64 бит, диапазон значений от $-9 \times 10^{18}$ до $+9 \times 10^{18}$.

При выборе формата необходимо учитывать, что, хотя сопроцессор и поддерживает целочисленные форматы, операции с ними выполняются недостаточно эффективно. Причина этого кроется в том, что все целочисленные данные переводятся в расширенный формат вещественного числа (внутреннее представление чисел в сопроцессоре), что требует дополнительных операций преобразования.

Числа в упакованном десятичном формате представлены в сопроцессоре в формате «9 байт + бит знака» (рис. 9.2).

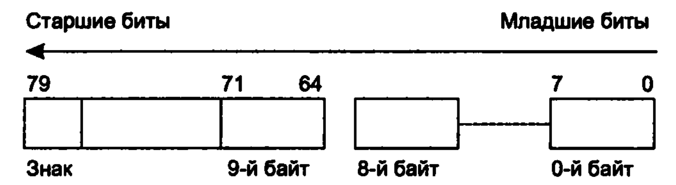



**Рис. 9.2.** Представление десятичного числа в сопроцессоре

Должен заметить, что сопроцессор включает только две команды для работы с упакованными десятичными числами.

Что же касается вещественных чисел, то они могут иметь размерность 32, 64 или 80 бит. Мы не будем подробно рассматривать способы внутреннего представления вещественных чисел в сопроцессоре, поскольку на практике это не имеет особого значения.

Диапазоны представления чисел в сопроцессоре следующие:

- короткое (32-разрядное) вещественное число — от $10^{-38}$ до $10^{+38}$;
- длинное вещественное число — от $10^{-308}$ до $10^{+308}$;
- расширенное вещественное число — от $10^{-4932}$ до $10^{+4932}$.

Ассемблерные программы чаще всего используют 32-разрядные вещественные числа. Для таких чисел обычной формой представления является двойное

слово, объявленное с директивой DD ассемблера. Длинные вещественные числа размерностью в 64 разряда могут быть представлены в виде учетверенного слова (директива DQ).

Специальные числовые или нечисловые значения могут получаться в результате выполнения математических операций сопроцессором. Например, к не-числам (Non-A-Number, NAN) относятся последовательности битов, которые нельзя сопоставить ни с одним форматом. К специальным числовым значениям относят ноль и бесконечность. Значение «ноль» может, например, быть результатом работы команд с нулевыми операндами. В перечень специальных числовых значений входят и так называемые денормализованные числа. Это числа, выходящие за пределы диапазона данного формата и приближающиеся к нулю.

## 9.2. Архитектура сопроцессора

Программная модель сопроцессора представляет собой набор дополнительных регистров, типов данных и команд. В группу регистров сопроцессора входят:

- восемь отдельно адресуемых 80-разрядных регистров, организованных в виде регистрового стека;
- три служебных 16-разрядных регистра: регистр состояния swr (Status Word Register) сопроцессора, управляющий регистр cwr (Control Word Register) и регистр тегов twr (Tags Word Register);
- регистры-указатели данных dpr (Data Point Register) и команд ipr (Instruction Point Register) используются при обработке исключительных ситуаций.

Все регистры доступны для программ с помощью специальных команд сопроцессора. Инструкции для обработки чисел с плавающей точкой тем или иным способом используют содержимое этих регистров. Рассмотрим более подробно регистры сопроцессора и начнем с регистрового стека. Его структура показана на рис. 9.3.

| | Знак | Порядок | Мантисса | | Регистр тегов | |
|---|---|---|---|---|---|---|
| | 79 | 78 64 | 63 0 | | 1 0 | |
| R0 | | | | ⟷ | | 1 — 0 |
| R1 | | | | ⟷ | | 3 — 2 |
| R2 | | | | ⟷ | | 5 — 4 |
| R3 | | | | ⟷ | | 7 — 6 |
| R4 | | | | ⟷ | | 9 — 8 |
| R5 | | | | ⟷ | | 11 — 10 |
| R6 | | | | ⟷ | | 13 — 12 |
| R7 | | | | ⟷ | | 15 — 14 |

**Рис. 9.3.** Регистровый стек сопроцессора

Каждый из восьми числовых регистров в стеке имеет размер 80 бит и состоит из отдельных полей, в соответствии с расширенным вещественным типом данных.

Команды сопроцессора адресуют регистры данных относительно регистра, являющегося вершиной стека. В любой момент времени номер этого регистра содержится в поле **top** регистра состояния (swr) сопроцессора.

Подобная организация стека и выбранный метод адресации (относительно вершины стека числовых регистров) упрощает программирование сопроцессора, поскольку позволяет подпрограммам передавать параметры через регистровый стек. Любая программа, выполняющая вычисления, может загрузить параметры в стек, после чего к ним можно получить доступ череэ логические имена регистров стека (st(0), st(1), st(2) и т. д.).

Регистр состояния (swr) содержит информацию о текущем состоянии сопроцессора. Некоторые наиболее важные для разработчика поля показаны на рис. 9.4:



**Рис. 9.4.** Регистр состояния (swr) сопроцессора

Поля **c3–c0** регистра устанавливаются определенным образом после выполнения математических операций и являются аналогом флагов в регистре флагов EFLAGS основного процессора. Поля 13–11 содержат текущий номер регистра стека (R0–R7), являющийся в данный момент вершиной стека.

Слово состояния регистра swr можно сохранить в памяти с помощью команд fstsw/fnstsw, fstenv/fnstenv и fsave/fnsave, после чего передать его в регистр AX с помощью команд fstsw AX/fnstsw AX, позволяя программе проверить слово состояния сопроцессора. Команда sahf может копировать поля **c3–c0** непосредственно в биты флагов процессора, что упрощает организацию ветвлений и условных переходов в программах.

Рассмотрим функционирование стека сопроцессора. Команды сопроцессора не обращаются напрямую к регистрам R0–R7, а используют так называемые логические номера регистров, имеющие обозначение st(0)–st(7). Регистровый стек имеет кольцевую организацию, а это означает, что вершина стека является плавающей и на нее указывает логический регистр st(0). Функционирование стека демонстрируют рис. 9.5 – 9.7. На рис. 9.5 показано состояние стека, когда регистр R5 является вершиной стека.

Операции загрузки данных в стек уменьшают значение **top** на единицу и загружают данные из памяти в новую вершину стека. Операции сохранения и восстановления данных из стека сохраняют значение из регистра, являющегося вершиной стека, в оперативную память, после чего увеличивают **top** на единицу. Подобно стеку основного процессора в памяти, регистровый стек сопроцессора растет вниз по направлению к регистрам с меньшими адресами. Если стек сопро-

цессора находится в состоянии, показанном на рис. 9.5, то при загрузке очередного числа вершина стека переместится к регистру R4 (рис. 9.6).

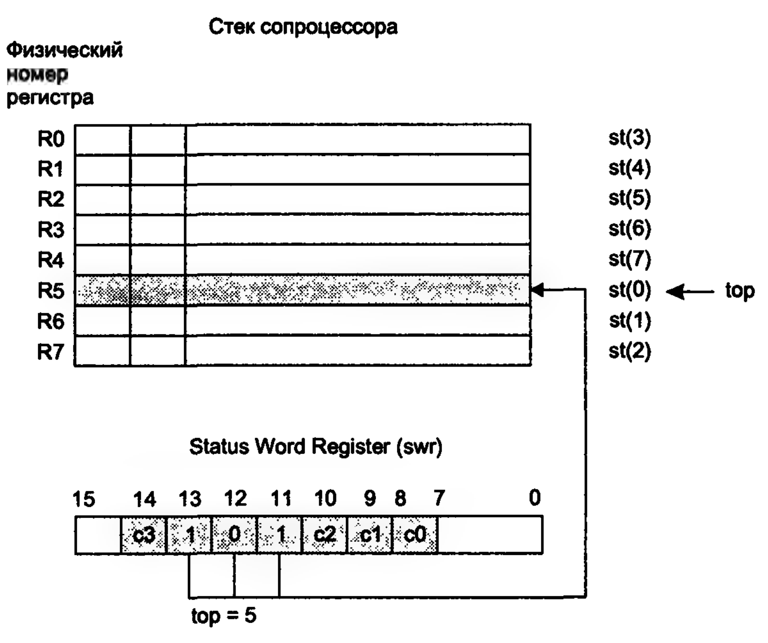



**Рис. 9.5.** Состояние стека, если вершина находится в физическом регистре R5

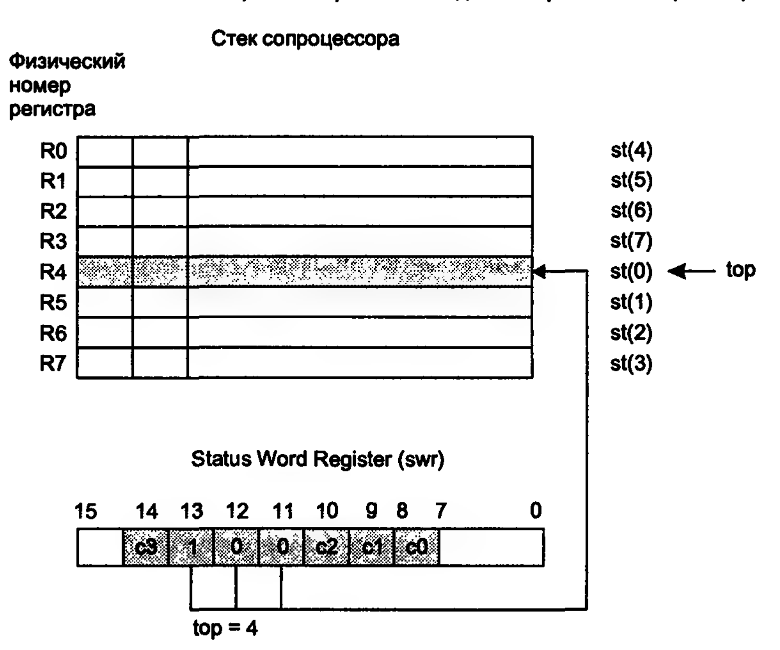



**Рис. 9.6.** Состояние стека при перемещении вершины стека в регистр R4

В этом случае биты 13–11 регистра состояния устанавливаются в значение, равное 4, как видно на рис. 9.6. Если теперь из стека сопроцессора удалить два числа, то вершина стека переместится к регистру R6, как показано на рис. 9.7.

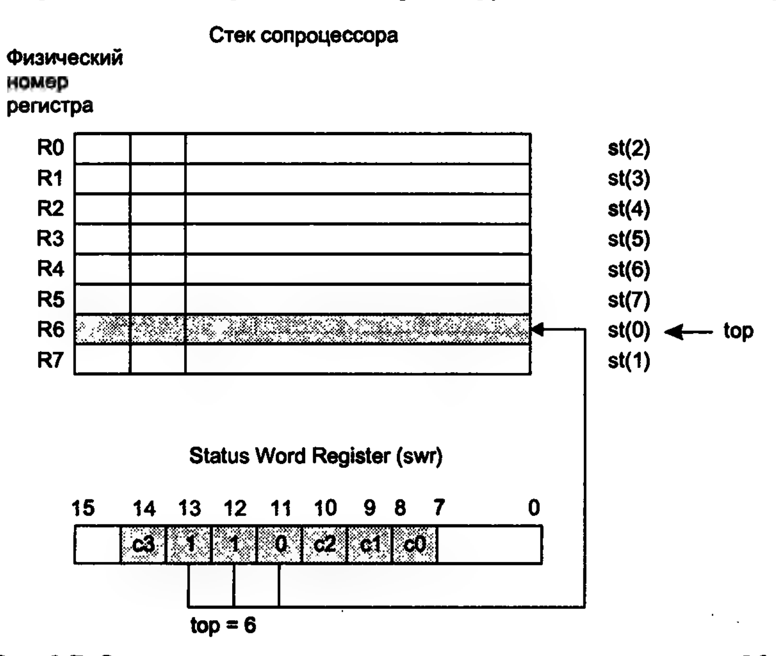



**Рис. 9.7.** Состояние стека при перемещении вершины стека в регистр R6

Соответствующие биты регистра состояния будут отображать состояние вершины стека и указывать номер регистра, являющегося вершиной стека (R6).

Для установки режимов обработки данных сопроцессора служит регистр управления (cwr). Он включает в себя шесть масок исключений, поля управления точностью (precision control, **pc**) и поля управления округлением (rounding control, **rc**). Формат регистра управления показан на рис. 9.8.

Вначале проанализируем смысл поля **pc**. С помощью битов этого поля выбирается длина мантиссы:

- 00 — длина мантиссы равна 24 бит;
- 10 — длина мантиссы равна 53 бит;
- 11 — длина мантиссы равна 64 бит.

Далее показаны возможные комбинации поля **rc** и соответствующие им режимы округления:

- 00 — число, содержащееся в стеке `st(0)` сопроцессора, округляется до ближайшего целого значения. Например, если требуется округлить число 3,14, то полученный результат равен 3. Если нужно округлить число 4,67, то полученный результат равен 5;

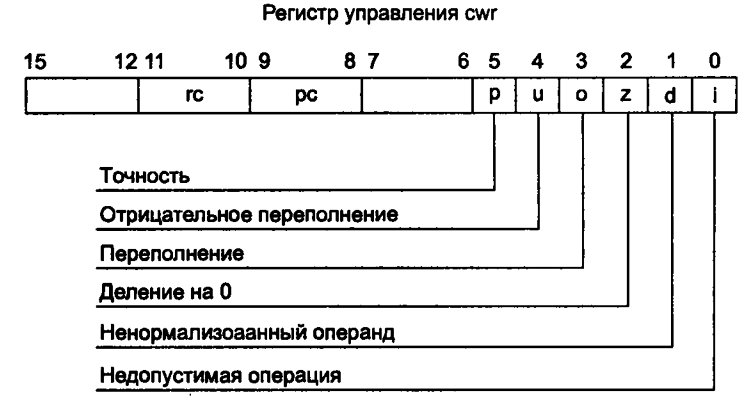



**Рис. 9.8.** Формат регистра управления cwr

- 01 — число в st(0) округляется в меньшую сторону. В этом случае для только что рассмотренных чисел (3,14 и 4,67) их округленные значения были бы равны 3 и 4 соответственно;
- 10 — число в st(0) округляется в большую сторону. В этом случае для только что рассмотренных чисел (3,14 и 4,67) их округленные значения были бы равны 4 и 5 соответственно;
- 11 — отбрасывается дробная часть числа. В этом случае число 3,14 округляется до 3, а 4,67 — до 4.

Шесть масок предназначены для маскирования исключительных ситуаций, возникновение которых фиксируется в регистре состояния. Если какие-то биты исключений в регистре cwr установлены в 1, это означает, что соответствующие исключения будут обрабатываться математическим сопроцессором. Если для какого-то типа исключения соответствующий бит маски cwr установлен в 1, то при возникновении исключения возбуждается прерывание. Обработчик прерывания должен либо находиться в самой операционной системе, либо разрабатываться программистом.

Когда сопроцессор обнаруживает ошибку, он пытается возбудить прерывание по особой ситуации, устанавливая соответствующие биты в слове состояния. Однако если ситуация замаскирована с помощью управляющего слова, сопроцессор сам реагирует на нее. Он решает, какая реакция соответствует данной ошибке, и возбуждает появление специфического числа в регистре. Например, результат NAN возникает, если операция некорректна, как в случае извлечения квадратного корня из отрицательного числа. Бесконечность появляется, если результат операции слишком велик для представления с плавающей точкой.

Биты управления точностью (биты 8–9) могут быть использованы для того, чтобы установить меньшую точность внутренних операций устройства, чем точность, заданная по умолчанию (размер мантиссы равен 64 бита). Биты управления точностью действуют только на результаты следующих арифметических операций: add, sub, mul, div и sqrt. Никакие другие операции этими битами не управляются.

Биты управления округлением (биты 10–11) предоставляют обычный режим округления до ближайшего целого, а также непосредственное округление и отсечение. Биты управления округлением действуют только на арифметические операции.

Рассмотрим еще один регистр сопроцессора — регистр тегов (twr). Этот регистр представляет собой совокупность двухразрядных полей. Каждое из этих полей соответствует определенному физическому регистру стека (см. рис. 9.2) и является индикатором состояния этого регистра. Если состояние одного из физических регистров R0–R7 изменяется, то это немедленно отражается на соответствующем этому регистру поле регистра тегов. Поле в регистре тегов может принимать одно из следующих значений:

- 00 — регистр стека сопроцессора содержит допустимое ненулевое значение;
- 01 — регистр стека сопроцессора содержит нулевое значение;
- 10 — регистр стека сопроцессора содержит одно из специальных численных значений (кроме нуля);
- 11 — регистр стека является пустым и допускает запись данных.

Если пытаться анализировать регистр тегов, то нужно принимать во внимание тот факт, что данные полей отображают состояние физических регистров R0–R7, а не логических st(0) – st(7). Для получения полной информации о состоянии регистров стека потребуется использовать поле **top** регистра состояния (swr).

# 9.3. Система команд математического сопроцессора

Команды с плавающей точкой можно сгруппировать в 5 функциональных классов:

- команды передачи данных;
- команды сравнения;
- трансцендентные команды;
- команды манипуляций константами;
- управляющие команды.

Обычно команда с плавающей точкой имеет один или два операнда, которые выбираются из регистрового стека сопроцессора или из памяти. Многие команды, такие, например, как fsin, по умолчанию работают с операндом в вершине регистрового стека. Другие команды требуют явной кодировки операнда или операндов в соответствии с мнемоникой операции. Существует еще один формат команд, принимающий один явный и один неявный операнд (обычно элемент в вершине стека).

Вне зависимости от того, определены операнды команд с плавающей точкой явно или принимаются по умолчанию, они делятся на два основных типа: операнд-источник и операнд-приемник. Даже когда команда преобразует операнд-источник из одного формата в другой (например, вещественное число в целое),

она работает во внутренней области, чтобы предотвратить изменение исходного операнда.

Многие команды позволяют кодировать операнды различными способами. Например, команда fadd (сложение чисел с плавающей точкой) может быть использована либо без операндов, либо только с операндом-источником, либо с операндом-источником и операндом-приемником. Когда указаны оба операнда, то операнд-приемник должен предшествовать операнду-источнику в командной строке и оба должны быть взяты из регистрового стека сопроцессора. Команды, обрабатывающие данные с плавающей точкой, либо выполняют чтение из памяти, либо сохраняют данные в памяти. При этом ни одна из этих команд не делает то и другое одновременно.

Перейдем теперь к анализу отдельных групп команд и начнем с команд передачи данных.

Команды передачи данных передают данные между регистрами стека и памятью. Эти команды можно разделить на три группы:

- команды передачи чисел с плавающей точкой;
- команды передачи целых чисел;
- команды передачи десятичных чисел.

Команды передачи данных автоматически обновляют регистр состояния сопроцессора. Рассмотрим команды передачи данных в формате вещественных чисел:

- fld *операнд-источник* — загружает вещественное число из ячейки памяти, представленной операндом-источником, в вершину стека сопроцессора;
- fst *операнд-приемник*, fstp *операнд-приемник* — обе команды сохраняют вещественное число из вершины стека в памяти. Единственное различие двух команд состоит в том, что команда fstp еще выталкивает операнд из стека, увеличивая тем самым значение **top** на 1. В этом случае вершиной стека становится физический регистр с большим номером.

Для передачи целочисленных данных используются следующие команды:

- fild *операнд-источник* — загружает число из ячейки памяти, представленной операндом-источником, в вершину стека;
- fist *операнд-приемник*, fistp *операнд-приемник* — обе команды сохраняют целое число из вершины стека в памяти. Единственное различие двух команд состоит в том, что команда fistp еще выталкивает операнд из стека, увеличивая тем самым значение **top** на 1. В этом случае вершиной стека становится физический регистр с большим номером.

Команды передачи данных, представленных в десятичном формате:

- fbld *операнд-источник* — загружает число из ячейки памяти, представленной операндом-источником, в вершину стека;
- fbstp *операнд-приемник* — сохраняет целое число из вершины стека в памяти и выталкивает операнд из стека, увеличивая тем самым значение top на 1. В этом случае вершиной стека становится физический регистр с большим номером.

К командам передачи можно отнести и группу команд для загрузки в стек регистров предопределенных констант:

- fldz — поместить 0 в вершину сопроцессора;
- fld1 — загрузка единицы в вершину стека сопроцессора;
- fldpi — загрузка числа π в вершину стека сопроцессора;
- fldl2t — загрузка двоичного логарифма десяти в вершину стека сопроцессора;
- fldl2e — загрузка двоичного логарифма числа е в вершину стека сопроцессора;
- fldlg2 — загрузка десятичного логарифма двух в вершину стека сопроцессора;
- fldln2 — загрузка натурального логарифма двух в вершину стека сопроцессора.

Существует еще одна команда, которую можно отнести к группе команд передачи данных, — это команда fxch st(*i*). Команда позволяет обменять значения в вершине стека сопроцессора и регистра st(*i*). Работу команд мы будем иллюстрировать примерами в виде 32-разрядных процедур, возвращающих результат в регистре EAX.

Перед использованием любых команд математического сопроцессора необходимо выполнить его инициализацию командой finit. Алгоритм работы команды выглядит следующим образом:

1. В регистр управления (cwr) помещается число 37h, означающее, что округление будет выполняться до ближайшего целого (поле **rc** = 0).
2. Биты с нулевого по пятый устанавливаются в 1, что означает маскирование всех исключений.
3. Поле управления точностью устанавливается для работы с максимальной точностью в 64 бит (**pc** = 11).
4. Регистр состояния (swr) обнуляется, а это означает, что исключения отсутствуют и физический регистр R0 регистрового стека становится вершиной стека st(0).
5. В регистр тегов (twr) заносятся единицы, а это означает, что все регистры сопроцессора считаются пустыми.
6. Регистры указателей данных (dpr) и команд (ipr) обнуляются.

Все вышеизложенное относится ко всем примерам этой главы, а начнем мы с копирования чисел с плавающей точкой (листинг 9.1).

**Листинг 9.1.** Копирование чисел с плавающей точкой

```
.686
.model flat
option casemap: none
.data
 src  DD 13.49, -71.01, -8.15, 33.39, 765.11
 len  EQU $-src
```

```
 dst  DD len DUP(0)
.code
 move_float proc
   finit
   mov  ECX, len          ; количество байтов массива src -> ECX
   shr  ECX, 2            ; привести к размерности двойного слова
   lea  ESI, src          ; адрес массива src -> ESI
   lea  EDI, dst          ; адрес массива dst -> EDI
next:
   fld  dword ptr [ESI]   ; поместить в вершину стека элемент массива src
   fstp dword ptr [EDI]   ; поместить в массив dst элемент
                          ; из вершины стека
   add  ESI, 4            ; перейти к следующему элементу в src
   add  EDI, 4            ; перейти к следующему элементу в dst
   dec  ECX               ; декремент счетчика
   jnz  next              ; если не равен 0, перейти
                          ; к следующей итерации
   lea  EAX, dst          ; вернуть в регистре EAX адрес массива dst
   ret
 move_float endp
 end
```

Процедура move_float выполняет копирование элементов массива src чисел с плавающей точкой в массив dst. Собственно копирование чисел выполняется командами fld и fstp. Обе команды оставляют вершину стека в том же состоянии, что и до вызова этих команд.

Процедура заполнения массива определенным значением показана в листинге 9.2

**Листинг 9.2.** Заполнение массива определенным значением

```
.686
.model flat
option casemap: none
.data
 src  DD 37 DUP (?)
 len  EQU $-src
 val  DD 1.45
.code
 set_value proc
   finit
   mov  ECX, len
   shr  ECX, 2
   lea  ESI, src
   fld  dword ptr val
next:
   fst  dword ptr [ESI]
   add  ESI, 4
   dec  ECX
   jnz  next
   fincstp
   lea  EAX, src
   ret
 set_value endp
 end
```

Процедура set_value выполняет заполнение массива src, содержащего 37 элементов, значением val, равным 1,45. Вначале значение val помещается в вершину стека с помощью команды

```
fld  dword ptr val
```

После этого в цикле выполняется копирование этой величины в элементы массива при помощи команды

```
fst  dword ptr [ESI]
```

Обратите внимание на то, что значение val не выталкивается из вершины стека до тех пор, пока цикл не закончится. После завершения всех итераций вершина стека очищается командой fincstp. Эта команда действует подобно команде fstp с той лишь разницей, что число выталкивается из стека в «никуда». Указатель вершины стека в поле **top** регистра swr после выполнения этой команды увеличивается на 1. Процедура возвращает адрес массива src в регистре EAX.

Листинг 9.3 демонстрирует, как выполняется команда fxch и функционирует стек регистров (процедура fxch_demo).

**Листинг 9.3.** Демонстрация использования регистров стека

```
.686
.model flat
option casemap: none
.data
memo  label qword
 memo1  DD 9.18       ; первое число
 memo2  DD 1.05       ; второе число
.code
 fxch_demo proc
   finit
   fld  memo1         ; поместить значение переменной
                      ; memo1 в стек
   fld  memo2         ; поместить memo2 в стек
   fxch st(1)         ; обменять значения в st(0) и st(1)
   fstp memo2         ; сохранить содержимое вершины стека в memo2
   fstp memo1         ; сохранить содержимое вершины стека в memo1
   lea  EAX, memo     ; вернуть адрес области памяти в регистре EAX
   ret
 fxch_demo endp
 end
```

Проанализируем работу процедуры fxch_demo. Пример может показаться достаточно сложным, поэтому для иллюстрации всех операций будем пользоваться рисунками. После выполнения следующей команды вершина стека будет содержать значение 9,18, как показано на рис. 9.9:

```
fld memo1
```

Следующая команда, fld memo2, загружает в стек регистров значение memo2. При этом вершина стека перемещается к физическому регистру с меньшим адресом и содержит значение 1,05. Значение переменной memo1, равное 9,18, перемещается в регистр st(1). Все это показано на рис. 9.10.




**Рис. 9.9.** Содержимое вершины стека после выполнения команды fld memo1

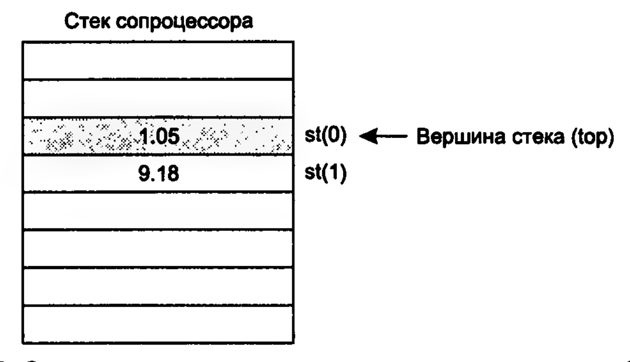



**Рис. 9.10.** Содержимое вершины стека после выполнения команды fld memo2

После выполнения следующей команды регистры `st(0)` и `st(1)` меняются содержимым (рис. 9.11):

```
fxch st(1)
```

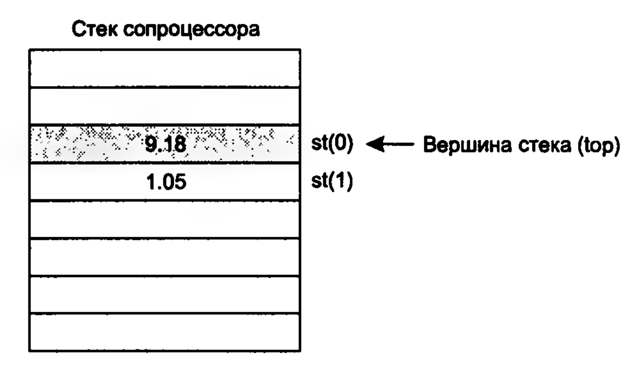



**Рис. 9.11.** Содержимое стека после выполнения команды fxch st(1)

Теперь остается сохранить содержимое стека в памяти. Первая команда, `fstp memo2`, сохраняет содержимое вершины стека в переменной `memo2`, при этом указатель вершины стека становится на 1 меньше (рис. 9.12).

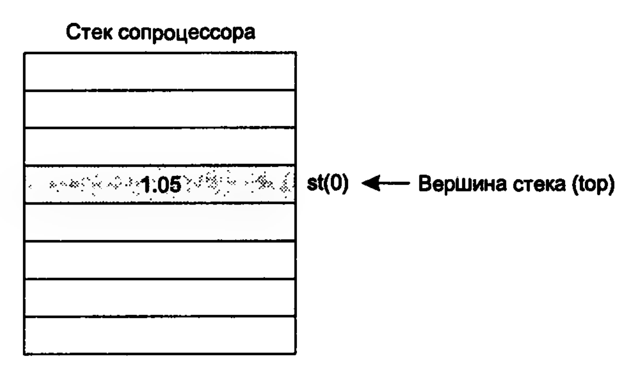



**Рис. 9.12.** Содержимое стека после выполнения команды fstp memo2

Наконец, после выполнения команды fstp memo1 содержимое стека (значение 1,05) помещается в переменную memo1. Таким образом происходит обмен значениями переменных memo1 и memo2. При этом стек регистров сопроцессора очищается.

Следующая группа команд, которую мы рассмотрим, — команды сравнения. Эти команды выполняют сравнение содержимого вершины стека с указанным в команде операндом. К этой группе относятся следующие команды:

- fcom — сравнение содержимого вершины стека st(0) с содержимым регистра st(1);
- fcomp mem — выполняет сравнение содержимого вершины стека st(0) с операндом памяти mem, после чего выталкивает число из st(0) и увеличивает указатель вершины стека на 1 в поле **top** регистра swr. Операнд mem может быть 32- или 64-разрядным числом с плавающей точкой;
- fcompp — выполняется так же, как и fcom, только обеспечивает выталкивание из стека значений, находящихся в регистрах st(0) и st(1);
- ficom — сравнение целочисленных значений st(0) и st(1);
- ficomp mem — выполняет сравнение содержимого вершины стека st(0) и операнда памяти mem, после чего выталкивает число из st(0) и увеличивает указатель вершины стека на 1 в поле **top** регистра swr. Операнд mem может быть 16- или 32-разрядным целым;
- ftst — сравнение содержимого вершины стека st(0) с нулем;
- fxam — выполняет анализ операнда, находящегося в вершине стека сопроцессора и по результату такого анализа устанавливает биты **c0—c3** специальным образом.

Команды сравнения дают правильный результат в тех случаях, когда операнды являются целыми числами или числами с плавающей точкой. Если только один из операндов является нечислом (NAN), то возбуждается исключительная ситуация и операция сравнения не производится.

Команды сравнения устанавливают биты кода условия в регистре swr, как показано в табл. 9.1.

**Таблица 9.1.** Коды условия команды fcom

| c3 | c2 | c1 | c0 | Значение |
|---|---|---|---|---|
| 0 | 0 | ? | 0 | st(0) больше операнда |
| 0 | 0 | ? | 1 | st(0) меньше операнда |
| 1 | 0 | ? | 0 | st(0) равен операнду |
| 1 | 1 | ? | 1 | st(0) и операнд несравнимы |

Манипулировать непосредственно битами кода условия **c3—c0** не совсем удобно. Существует другой механизм, с помощью которого можно определить результат сравнения привычным способом, используя битовые флаги регистра флагов процессора. Для этих целей в состав системы команд сопроцессора включена команда fstsw, позволяющая сохранить слово состояния в регистре AX или в ячейке памяти. Интересующие нас биты легко могут быть извлечены с помощью команды sahf, которая записывает содержимое регистра AH в младший байт регистра флагов.

При этом бит **c0** помещается на место флага CF, **c2** – на место флага PF, **c3** – на место ZF, а бит **c1** не используется. Рассмотрим несколько примеров выполнения команд сравнения, начав с поиска большего из двух чисел (листинг 9.4).

**Листинг 9.4.** Поиск большего из двух чисел

```
.686
.model flat
option casemap: none
.data
 a1  DD -162.31
 a2  DD -117.03
 res DD 0
.code
max proc
  finit
  fld   dword ptr a1
  fld   dword ptr a2
  fcom
  fstsw AX
  sahf
  jnc   store
  fxch  st(1)
store:
  fstp  res
  fwait
  lea   EAX, res
  ret
 max endp
 end
```

Процедура max, исходный текст которой приведен в листинге 9.4, сравнивает два числа с плавающей точкой и в качестве результата в регистре EAX возвращает значение большего из них. Вначале в стек регистров сопроцессора загружается

первое число, затем — второе. Для сравнения двух чисел используется команда fcom без параметров, предполагающая наличие операндов в стеке. К моменту выполнения команды fcom в вершине стека находится число a2, в то время как число a1 находится в регистре st(1).

После сравнения флаги сохраняются в регистре AX с помощью команды fstsw AX, а следующая команда, sahf, извлекает их в регистр флагов EFLAGS. Здесь следует быть очень внимательным. Для анализа результата сравнения можно использовать три значащих флага:

- CF — соответствует биту **c0**;
- ZF — соответствует биту **c3**;
- PF — соответствует биту **c2**.

Различным комбинациям этих флагов соответствуют различные команды условного перехода. Например, условием того, что операнд в регистре st(0) больше операнда в st(1), является равенство нулю флага переноса (CF = 0). В этом случае для ветвления программы можно использовать команду jnc store, при этом число в вершине стека является максимумом и сохраняется в переменной res. В противном случае регистры st(0) и st(1) меняются операндами и в вершине стека оказывается большее число, которое и сохраняется в переменной res.

Возможен и другой вариант — использовать мнемонически более понятную команду ja store. Эта команда анализирует флаг CF на равенство нулю, поэтому фрагмент кода, где присутствует команда jnc, можно переписать следующим образом:

```
. . .
fcom
  fstsw AX
  sahf
  ja   store
  fxch st(1)
store:
. . .
```

При указанных значениях операндов переменная res будет содержать значение, равное –117,03.

В листинге 9.5 показана процедура сравнения массивов, содержащих числа с плавающей точкой.

**Листинг 9.5.** Сравнение массивов, содержащих числа с плавающей точкой

```
.686
.model flat
option casemap: none
.data
 a1  DD -117.03, 8.04, -9.21, 5.16
 a2  DD -117.03, 8.04, -9.21, 5.16
 len EQU $-a2
 equals DB "Arrays are equal!", 0
 non_eq DB "Arrays are not equal!",0
```

```
.code
arrays_eq proc
  lea   ESI, a1
  lea   EDI, a2
  mov   ECX, len
  shr   ECX, 2
  finit
next:
  fld   dword ptr [ESI]
  fld   dword ptr [EDI]
  fcom
  fstsw AX
  sahf
  jne   n_eq
  add   ESI, 4
  add   EDI, 4
  dec   ECX
  jnz   next
  lea   EAX, equals
  jmp   exit
n_eq:
  lea   EAX, non_eq
exit:
  fwait
  ret
arrays_eq endp
end
```

Процедура arrays_eq в этом примере сравнивает два массива (a1 и a2) чисел с плавающей точкой одинакового размера. В случае равенства массивов в регистре EAX возвращается строка equals, в случае неравенства — строка n_eq. Перед началом операции в регистры ESI и EDI загружаются адреса массивов a1 и a2, а в регистр ECX — размер массивов a1 и a2. Элементы массивов загружаются в вершину стека сопроцессора с помощью команд fld, после чего выполняется операция сравнения элементов командой fcom.

Как и в предыдущем примере, для обработки результата сравнения выполняются команды:

```
fstsw AX
sahf
jne   n_eq
```

Равенство или неравенство элементов определяется флагом ZF, поэтому используется команда jne. Если до окончания цикла сравнения какие-либо элементы оказываются неравными, то происходит выход из цикла по команде jne n_eq.

В группу команд сравнения включена очень полезная команда — ftst, позволяющая сравнить содержимое вершины стека st(0) с нулем. Следующий пример демонстрирует применение этой команды. Процедура sort_zero ищет в целочисленном массиве a1 чисел с плавающей точкой ненулевые элементы и сохраняет их в другом массиве — a2. Исходный текст процедуры показан в листинге 9.6.

**Листинг 9.6.** Поиск ненулевых элементов массива чисел с плавающей точкой

```
.686
.model flat
option casemap: none
.data
 a1  DD -17.03, 0, 9.21, 0, -67.3
 len EQU $-a1
 a2  DD  5 DUP (0)
.code
sort_zero proc
  lea   ESI, a1
  lea   EDI, a2
  mov   ECX, len
  shr   ECX, 2
  finit
next:
  fld   dword ptr [ESI]
  ftst
  fstsw AX
  sahf
  je    skip
  fstp  dword ptr [EDI]
  add   EDI, 4
skip:
  add   ESI, 4
  dec   ECX
  jnz   next
  fwait
  lea   EAX, a2
  ret
sort_zero endp
end
```

Программный код процедуры sort_zero во многом напоминает предыдущие примеры, поэтому остановлюсь лишь на основных различиях. Следующая группа команд анализирует содержимое вершины стека на равенство нулю (ZF = 1) и либо пропускает такой элемент, если он нулевой (переход на метку skip), либо сохраняет его в другом массиве (в данном случае — a2):

```
ftst
fstsw  AX
sahf
je    skip
```

Массивы элементов адресуются регистрами ESI (исходный массив a1) и EDI (массив a2 ненулевых элементов). Размер массива a2 в общем случае должен быть не меньше размера массива a1.

Следующая команда, которую мы рассмотрим, — fxam. Команда очень полезна при анализе содержимого вершины стека st(0), что часто помогает при отладке программ. В зависимости от результата анализа команда fxam устанавливает соответствующим образом биты **c0—c3** в регистре состояния сопроцессора swr. Некоторые из комбинаций этих регистров и соответствующие им состояния приведены в табл. 9.2.

Рассмотрим пример использования команды fxam. Исходный текст процедуры (она называется _fxam_demo) показан в листинге 9.7.

**Таблица 9.2.** Биты состояния и содержимое вершины стека

| c3 | c2 | c1 | c0 | Значение |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | Положительное нечисло (+NAN) |
| 0 | 0 | 1 | 1 | Отрицательное нечисло (–NAN) |
| 0 | 1 | 0 | 0 | Корректное положительное число |
| 0 | 1 | 0 | 1 | Положительная бесконечность |
| 0 | 1 | 1 | 0 | Корректное отрицательное число |
| 0 | 1 | 1 | 1 | Отрицательная бесконечность |

**Листинг 9.7.** Определение типа содержимого вершины стека

```
.686
.model flat
option casemap: none
.data
  a1       DD 45.2, -3.14, -88.4, 5.6, -11.34, 0, 1.33
  len      EQU $-op1
  pos_num  DB "Number is positive",0
  neg_num  DB "Number is negative",0
  other    DB "Other meaning", 0
  err_parm DB "Incorrect value of parameter!",0
.code
_fxam_demo proc
  push  EBP
  mov   EBP, ESP
  mov   ECX, dword ptr [EBP+8]
  cmp   ECX, 6
  jg    wrong_param
  shl   ECX, 2
  lea   ESI, a1
  finit
  fld   dword ptr [ESI][ECX]
  fxam
  fstsw AX
  and   AH, 7h
  cmp   AH, 4h
  je    pos_correct
  cmp   AH, 6h
  je    neg_correct
  lea   EAX, other
  jmp   exit
pos_correct:
  lea   EAX, pos_num
  jmp   exit
neg_correct:
  lea   EAX, neg_num
  jmp   exit
wrong_param:
  lea   EAX, err_parm
exit:
  pop   EBP
  ret
_fxam_demo endp
end
```

Процедура _fxam_demo передает в вызывающую программу строку с информацией об элементе массива a1 чисел с плавающей точкой, принимая в качестве параметра номер элемента массива. Параметр извлекается с использованием регистра EBP и помещается в ECX, причем его значение должно находиться в диапазоне 0–6, что отслеживается командой

```
cmp ECX, 6
```

При выходе значения счетчика за пределы диапазона в регистр EAX помещается адрес строки err_parm и процедура завершается.

Адрес массива a1 загружается в регистр ESI с помощью команды

```
lea ESI, a1
```

После этого элемент массива помещается в вершину стека и выполняется его анализ:

```
fld   dword ptr [ESI][ECX]
fxam
```

Далее выполняется анализ битов **c0–c3** регистра состояния (swr), для чего содержимое регистра swr вначале помещается в EAX командой fstsw AX. В данной процедуре мы анализируем три состояния вершины стека:

- является ли содержимое вершины стека корректным значением положительного числа;
- является ли содержимое вершины стека корректным значением отрицательного числа;
- является ли содержимое вершины стека чем-то иным, чем указанные выше значения.

Для удобства анализа замаскируем незначащие для процедуры биты в регистре AH:

```
and   AH, 7h
```

Согласно табл. 9.2 корректному положительному числу соответствует значение 4 (**c2** = 1, **c1** = 0, **c0** = 0), поэтому можно проверить этот вариант с помощью команд

```
cmp   AH, 4h
je    pos_correct
```

Если число является корректным и положительным, в регистр EAX помещается адрес строки pos_num и происходит выход из процедуры. Если это не так, выполняется проверка на корректное отрицательное число:

```
cmp   AH, 6h
je    neg_correct
```

Если содержимое вершины стека является корректным отрицательным числом, то в регистр EAX помещается адрес строки neg_num и процедура завершается. Если не выполняется ни одно из предыдущих двух условий, то в регистр EAX заносится адрес строки other и происходит выход из процедуры.

В приводимых примерах использовались только команды для обработки чисел с плавающей точкой, но модифицировать эти примеры так, чтобы они работали с целыми числами, не составит труда. Для этого команды обработки чисел с плавающей точкой следует заменить соответствующими командами обработки целых чисел.

Начиная с процессора Intel Pentium Pro, группа команд сравнения пополнилась двумя новыми командами: fcomi и fcomip. Эти команды выполняют те же операции, что и fcom и fcomp, но в отличие от последних не требуют специальных команд для обработки битов **c0—c3** регистра swr, а устанавливают флаги ZF, CF и PF непосредственно в регистре флагов EFLAGS центрального процессора. Рассмотрим пример применения команды fcomi. Следующая процедура (она называется fcomi_demo) сравнивает два числа с плавающей точкой, которые являются ее параметрами, и возвращает адрес большего из них в регистре EAX (листинг 9.8).

**Листинг 9.8.** Сравнение двух чисел с плавающей точкой

```
.686
.model flat
option casemap:none
.data
 res DD 0
.code
fcomi_demo proc
  push  EBP
  mov   EBP, ESP
  finit
  fld   dword ptr [EBP+12]
  fld   dword ptr [EBP+8]
  fcomi st, st(1)
  ja    save_op1
  fxch  st(1)
save_op1:
  fstp  dword ptr res
  lea   EAX, res
  pop   EBP
  ret
fcomi_demo endp
end
```

Для извлечения операндов процедура использует регистр EBP. Оба операнда загружаются в стек регистров при помощи двух команд fld. Команда fcomi st, st(1) сравнивает содержимое регистров st(0) и st(1) и в зависимости от результата устанавливает флаги ZF, CF и PF. Большее из чисел помещается в регистр st(0) после выполнения команд

```
  ja    save_op1
  fxch  st(1)
save_op1:
  fstp  dword ptr res
```

Команда fxch st(1) выполняет обмен содержимым регистров st(0) и st(1), если содержимое st(1) больше содержимого st(0). Значение максимума помещается в переменную res, а адрес переменной res — в регистр EAX.

Использование команды fcomi позволяет избавиться от команд fstsw AX и sahf, что повышает (особенно при большом количестве операций сравнения, выполняемых в циклах) быстродействие программы.

Помимо команд fcomi и fcomip, в систему команд процессоров Intel, начиная с Pentium Pro, включены команды, которые можно обозначить как fcmov*CC*, где *CC* — мнемоническое обозначение кода условия (b, nb, e, ne, be, nbe). Эти команды выполняют копирование регистра st(*i*) в регистр st(0), если выполняется указанное в команде условие. Форматы команд и анализируемые ими флаги показаны в табл. 9.3.

**Таблица 9.3.** Форматы команд fcmovCC

| Мнемоническое обозначение | Состояние флагов | Описание условия |
|---|---|---|
| fcmovb | CF = 1 | Меньше |
| fcmovnb | CF = 0 | Не меньше |
| fcmove | ZF = 1 | Равно |
| fcmovne | ZF = 0 | Не равно |
| fcmovbe | (CF или ZF) = 1 | Меньше или равно |
| fcmovnbe | (CF или ZF) = 0 | Не меньше или не равно |

Так же как и команды cmov*CC*, команды fcmov*CC* хорошо подходят для оптимизации вычислительных алгоритмов, исключая избыточные ветвления и переходы в программе. Хочу отметить, что не все процессоры семейства Pentium Pro поддерживают эти команды, поэтому, прежде чем их применять, нужно получить расширенную информацию о процессоре с помощью команды cpuid.

Продемонстрирую возможности команд fcmov*CC* на примере программного кода процедуры fcmov_ex (листинг 9.9).

**Листинг 9.9.** Попарное сравнение элементов массивов

```
.686
.model flat
option casemap: none
.data
  a1  DD  56.78. -43.2. 0.0. 78.23. -12.2
  len EQU $-a1
  b1  DD  134.78. 67.45. -8.5. 32.18. -17.04
  res DD  len DUP (0)
.code
 fcmov_ex proc
   mov  ECX. len
   shr  ECX. 2
   lea  ESI. a1
   lea  EDI. b1
   lea  EDX. res
   finit
 next:
   fld  dword ptr [ESI]
   fld  dword ptr [EDI]
```

```
  fcomi st. st(1)
  fcmovb st,st(1)
  fstp dword ptr [EDX]
  add  ESI. 4
  add  EDI. 4
  add  EDX. 4
  dec  ECX
  jnz  next
  lea  EAX. res
  ret
 fcmov_ex endp
 end
```

В области данных процедуры определены три массива чисел с плавающей точкой: a1, b1 и res. Программа выполняет попарное сравнение элементов массивов a1 и b1, находящихся на одинаковых позициях, и помещает больший из них в массив res. Процедура возвращает адрес массива res в регистре EAX. Проанализируем программный код процедуры.

Доступ к элементам всех трех массивов осуществляется посредством регистров ESI, EDI и EDX, а счетчик элементов массивов находится в регистре ECX. В каждой итерации в регистры стека st(0) и st(1) загружаются элементы массивов a1 и b1 (команды fld dword ptr [ESI] и fld dword ptr [EDI]). С помощью команды fcomi st. st(1) числа сравниваются, и в результате устанавливаются соответствующие биты в регистре флагов.

Команда fcmovb st. st(1) копирует содержимое регистра st(1) в регистр st(0) в случае, если значение в st(0) оказалось меньше st(1). Если условие не выполнено, то есть содержимое st(0) больше st(1), команда ничего не делает. Таким образом, в регистре, являющемся вершиной стека, в любом случае окажется больший элемент. Применение команды fcmovb позволяет при этом избежать ветвления с помощью одной из команд перехода.

Команда fstp dword ptr [EDX] запоминает содержимое регистра st(0) в массиве res, а инструкция lea EAX. res сохраняет адрес массива в регистре EAX, после чего осуществляется переход к адресам следующих элементов массивов:

```
add  ESI. 4
add  EDI. 4
add  EDX. 4
```

Команда dec ECX управляет числом итераций цикла, выполняя переход к следующей итерации, если содержимое ECX не равно 0.

Прежде чем закончить анализ команд сравнения, хочу упомянуть о ситуации, когда один или оба операнда оказываются некорректными целыми или вещественными числами. Для того чтобы все-таки выполнить операции над такими числами, в системе команд предусмотрены три команды:

- fucom st(*i*) — сравнение неупорядоченных чисел в регистрах st(0) и st(*i*);
- fucomp st(*i*) — сравнение неупорядоченных чисел в регистрах st(0) и st(*i*) с выталкиванием чисел из вершины стека;
- fucompp st(*i*) — сравнение неупорядоченных чисел в регистрах st(0) и st(*i*) с выталкиванием чисел из регистров st(0) и st(1).

В группу арифметических команд входят команды, реализующие операции сложения, вычитания, умножения и деления. В свою очередь, арифметические команды можно разделить на две подгруппы для работы с целочисленными и вещественными операндами. Вначале рассмотрим команды для работы с целочисленными операндами. Они манипулируют данными размером в 16 и 32 бита и включают в себя:

- fiadd *операнд* — сложение содержимого вершины стека st(0) с целочисленным операндом размером 16 или 32 бита, результат сложения сохраняется в регистре st(0);
- fisub *операнд* — вычитание целочисленного операнда размером 16 или 32 бита из содержимого вершины стека st(0), результат вычитания сохраняется в регистре st(0);
- fimul *операнд* — умножение содержимого вершины стека st(0) на целочисленный операнд размером 16 или 32 бита, результат умножения сохраняется в регистре st(0);
- fdiv *операнд* — деление содержимого вершины стека st(0) на целочисленный операнд размером 16 или 32 бита, результат деления сохраняется в регистре st(0);
- fisubr *операнд* — вычитание содержимого регистра st(0) из целочисленного операнда размером 16 или 32 бита, результат вычитания сохраняется в регистре st(0);
- fidivr *операнд* — деление целочисленного операнда размером 16 или 32 бита на содержимое вершины стека st(0), результат деления сохраняется в регистре st(0).

Приведу пример использования целочисленных арифметических команд. Предположим, необходимо вычислить выражение $(a + b)/(a - b)$, где $a$ и $b$ — целые числа. Эту операцию можно выполнить с помощью процедуры _int_ops, мнемоническое обозначение которой можно представить как _int_ops(*a*, *b*). Процедура возвращает округленное до ближайшего целого числа значение, если параметры $a$ и $b$ не равны, или 0 в случае равенства операндов. Исходный текст процедуры показан в листинге 9.10.

**Листинг 9.10.** Вычисление выражения, содержащего целые числа

```
.686
.model flat
option casemap: none
.data
  op1  DD 0
  res  DD 0
.code
_int_ops proc
  push   EBP
  mov    EBP, ESP
  finit
  fild   dword ptr [EBP+8]
  ficom  dword ptr [EBP+12]
```

```
  fstsw  AX
  sahf
  jz     eq_0
  fisub  dword ptr [EBP+12]
  fistp  dword ptr opl
  fild   dword ptr [EBP+8]
  fiadd  dword ptr [EBP+12]
  fidiv  dword ptr opl
  fistp  dword ptr res
eq_0:
  lea    EAX. res
exit:
  pop    EBP
  ret
_int_ops endp
end
```

Предположим, что параметры в процедуру передаются через стек, причем по большему адресу в стеке находится число *b*, а по меньшему — число *a*. Для извлечения параметров из стека используется регистр EBP, тогда число *a* находится по адресу [EBP+8], а число *b* — по адресу [EBP+12]. Поскольку делитель выражения, равный *a* – *b*, не должен равняться нулю, то вначале выполняется проверка на равенство значений переменных *a* и *b*:

```
fild   dword ptr [EBP+B]
ficom  dword ptr [EBP+12]
fstsw  AX
sahf
jz     eq_0
```

Если *a* равно *b*, то происходит выход из процедуры, а в регистре EAX возвращается 0. Если же операнды не равны, то вначале находится разность *a* – *b* (делитель), которая сохраняется в переменной opl:

```
fisub  dword ptr [EBP+12]
fistp  dword ptr op2
```

Далее необходимо найти сумму *a* + *b*, что делается при помощи команд

```
fild   dword ptr [EBP+8]
fiadd  dword ptr [EBP+12]
```

Наконец, команда fidiv dword ptr opl находит частное (*a* + *b*)/(*a* – *b*), которое затем помещается в переменную res. Поскольку выполняется целочисленное деление, то результат округляется до ближайшего целого числа.

Большим разнообразием выделяется набор арифметических команд для работы с вещественными числами. Анализ начнем с команд сложения. Команды этой группы имеют следующий синтаксис:

```
fadd  приемник. источник
faddp приемник. ST
fadd  источник
```

Здесь *приемник* — операнд-приемник, а *источник* — операнд-источник. Команды выполняют сложение содержимого операнда-источника с содержимым

операнда-приемника, сохраняя результат в операнде-приемнике. Если оба операнда являются регистрами стека, то одним из них обязательно должен быть st(0). Если единственным операндом является ячейка памяти (32 или 64 разряда), то вторым операндом будет регистр st(0), в котором и сохраняется результат.

Далее приводится более подробное описание команд сложения:

- fadd — сложение содержимого вершины стека st(0) и регистра st(1), результат сложения сохраняется в регистре st(0);
- fadd *источник* — сложение содержимого вершины стека st(0) и ячейки памяти *источник*, результат сложения сохраняется в регистре st(0);
- fadd st, st(*i*) — сложение содержимого регистра стека st(*i*) и вершины стека st(0), результат сохраняется в регистре st(0);
- fadd st(*i*), st — сложение содержимого регистра стека st(*i*) и вершины стека st(0), результат сохраняется в регистре st(*i*);
- faddp st(*i*), st — выполняет сложение содержимого регистра стека st(*i*) и вершины стека st(0), после чего выталкивает значение из вершины стека, результат операции сохраняется в регистре st(*i*-1).

Следующая подгруппа команд, которую мы проанализируем, — команды вычитания вещественных чисел. Команды этой группы имеют следующий синтаксис:

```
fsub  приенник, источник
fsubp приемник, ST
fsub  источник
```

Здесь *приемник* — операнд-приемник, а *источник* — операнд-источник. Команды выполняют вычитание содержимого операнда-источника из содержимого операнда-приемника, сохраняя результат в операнде-приемнике. Если оба операнда являются регистрами стека, то одним из них обязательно должен быть st(0). Если единственным операндом является ячейка памяти (32 или 64 разряда), то вторым операндом будет регистр st(0), в котором и сохраняется результат.

Вычитание чисел с плавающей точкой выполняется такими командами:

- fsub — вычитание содержимого регистра st(1) из содержимого вершины стека st(0), результат вычитания сохраняется в регистре st(0);
- fsub *источник* — вычитание содержимого операнда *источник*, расположенного в памяти, из содержимого вершины стека st(0), результат вычитания сохраняется в регистре st(0);
- fsub st, st(*i*) — вычитание содержимого регистра st(*i*) из содержимого регистра стека st(0), результат сохраняется в регистре st(0);
- fsub st(*i*), st — вычитание содержимого регистра стека st(0) из содержимого любого из регистров st(*i*), результат сохраняется в регистре st(0);
- fsubp st(*i*), st — выполняет вычитание содержимого регистра стека st(0) из содержимого любого из регистров st(*i*), после чего выталкивает значение из вершины стека, результат операции сохраняется в регистре st(*i*-1);
- fsubr st(*i*), st — вычитание содержимого регистра стека st(0) из содержимого любого из регистров st(*i*), результат сохраняется в регистре st(0);

- fsubrp st(*i*), st — выполняет вычитание содержимого регистра стека st(0) из содержимого любого из регистров st(*i*), после чего выталкивает значение из вершины стека, результат операции сохраняется в регистре st(*i*-1).

Следующая подгруппа команд, которые мы будем анализировать, — команды умножения. Все команды этой группы имеют следующий синтаксис:

```
fmul  приемник, источник
fmulp приемник, ST
fmul  источник
```

Здесь *приемник* — операнд-приемник, а *источник* — операнд-источник. Команды этой группы выполняют умножение операнда-приемника на операнд-источник, сохраняя результат в операнде-приемнике. Если в качестве операндов указаны регистры стека сопроцессора, то один из них должен быть st(0). Если в качестве единственного операнда выступает ячейка памяти, то вторым операндом по умолчанию является регистр st(0), сохраняющий результат операции. Операнды, представленные переменными в памяти, могут быть 32- или 64-разрядными. В эту группу включены следующие команды:

- fmul — умножение содержимого регистра вершины стека st(0) на содержимое регистра st(1), результат сохраняется в регистре st(0);
- fmul st, st(*i*) — умножение регистра st(0) на содержимое регистра st(*i*), результат сохраняется в регистре st(0);
- fmul st(*i*), st — умножение содержимого регистра st(0) на содержимое одного из регистров st(*i*), результат сохраняется в регистре st(*i*);
- fmulp st(*i*), st — выполняет умножение, так же как команда fmul st(*i*), st, и, кроме того, выталкивает содержимое вершины стека st(0), результат умножения остается в регистре st(*i*-1).

Последняя подгруппа команд, которую мы рассмотрим, — команды деления. Все команды этой группы имеют следующий синтаксис:

```
fdiv приемник, источник
fdivp приемник, ST
fdiv источник
```

Здесь *приемник* — операнд-приемник, а *источник* — операнд-источник. Команды этой группы выполняют деление операнда-приемника на операнд-источник, сохраняя частное в операнде-приемнике. Если в качестве операндов указаны регистры стека сопроцессора, то один из них должен быть st(0). Если в качестве единственного операнда выступает ячейка памяти, то вторым операндом по умолчанию является регистр st(0), в котором сохраняется результат операции. Операнды, представленные переменными в памяти, могут быть 32- или 64-разрядными. В эту группу включены такие команды:

- fdiv — деление содержимого регистра st(1) на значение, находящееся в вершине стека st(0), результат операции сохраняется в регистре st(0);
- fdiv st, st(*i*) — деление содержимого регистра st(0) на значение, находящееся в регистре st(*i*), результат операции сохраняется в регистре st(0);

- fdiv st(*i*), st — деление содержимого регистра st(*i*) на значение, находящееся в регистре st(0), результат операции сохраняется в регистре st(*i*);
- fdivp st(*i*), st — выполняет деление, так же как и команда fdiv st(*i*), st, но выталкивает содержимое вершины стека, результат операции помещается в регистр st(*i*-1);
- fdivr st(*i*), st — деление содержимого регистра st(*i*) на значение, находящееся в вершине стека st(0), результат операции помещается в регистр st(0);
- fdivrp st(*i*), st — деление содержимого регистра st(*i*) на значение, находящееся в вершине стека st(0), результат операции помещается в регистр st(*i*), после чего содержимое st(0) выталкивается из стека, а результат деления остается в регистре st(*i*-1).

Для демонстрации работы вещественных арифметических команд разработаем простую процедуру (назовем ее real_ops), в которой вычисляется значение выражения $c_1 \times (a_1 + b_1)/d_1 \times (a_1 - b_1)$, где $a_1$, $b_1$, $c_1$, $d_1$ — числа с плавающей точкой (листинг 9.11).

**Листинг 9.11.** Вычисление значения выражения, содержащего числа с плавающей точкой

```
.686
.model flat
option casemap: none
.data
  a1   DD 2.0
  b1   DD 5.5
  c1   DD -3.5
  d1   DD 6.8
  res  DD 0
.code
 real_ops proc
   finit              ; инициализировать сопроцессор
   fld  dword ptr a1  ; поместить a1 в вершину стека st(0)
   fcom dword ptr b1  ; сравнить с b1, если равны, то знаменатель
                      ; выражения равен 0
                      ; выйти из процедуры, поместив в регистр
                      ; значение 0
   fstsw AX           ; сохранить содержимое регистра состояния swr
                      ; в регистре AX
   sahf               ; поместить содержимое регистра AH
                      ; в регистр флагов
   jz   exit          ; если a1 = b1, выйти из процедуры, установив
                      ; регистр AX в 0, иначе продолжить
                      ; вычисления
   fadd dword ptr b1  ; прибавить к a1 значение b1, после чего
                      ; вершина стека st(0) будет содержать a1 + b1
   fmul dword ptr c1  ; умножить содержимое st(0) на c1
                      ; после этой операции вершина стека st(0)
                      ; содержит значение c1 * (a1 + b1)
   fld  dword ptr a1  ; загрузить в стек значение a1
   fsub dword ptr b1  ; вычесть b1 из a1
   fmul dword ptr d1  ; умножить разность b1 - a1 на d1
                      ; после этой операции вершина стека st(0)
```

```
                         ; содержит значение d1 * (a1 - b1). Значение
                         ; c1 * (a1 + b1) находится в регистре st(1)
  fdiv                   ; разделить содержимое регистра st(1)
                         ; на содержимое вершины стека st(0)
  fstp dword ptr res     ; сохранить значение c1 * (a1 + b1)/d1 * (a1 - b1)
                         ; в переменной res и вытолкнуть значение
                         ; из вершины стека
exit:
  lea  EAX, res          ; поместить адрес результата в регистр EAX
  ret
real_ops endp
end
```

При указанных значениях операндов перед выходом из процедуры переменная res будет содержать значение 1,1.

Для вычисления значений тригонометрических функций, таких, как синус, косинус, тангенс, арктангенс, а также логарифмических и показательных функций в математический сопроцессор включен целый ряд так называемых трансцендентных команд. С их помощью выполняются вычисления для всех обычных тригонометрических, обратных тригонометрических, гиперболических, обратных гиперболических, логарифмических и степенных функций. Обычно такие вычисления занимают много времени и являются очень интенсивными по использованию ресурсов процессора.

Трансцендентные команды, как правило, работают с верхними элементами регистрового стека — регистром вершины стека st(0) и регистром st(1). Команды для вычисления тригонометрических функций работают с аргументами, выраженными в радианах, поэтому для вычисления функции угла, заданного в градусах, следует вначале преобразовать это значение в радианы. Для такого преобразования можно использовать формулу

$$\text{Угол (рад)} = \text{Угол (град)} \times \pi/180.$$

Рассмотрим более подробно трансцендентные команды. Вначале остановимся на командах вычисления синуса и косинуса, которые имеют такой синтаксис:

```
fsin
fcos
fsincos
```

Команда fsin замещает аргумент (в радианах), расположенный в вершине стека st(0), значением синуса. Аналогично, команда fcos вычисляет косинус угла и помещает его в st(0). Команда fsincos комбинирует предыдущие две команды и вычисляет обе функции следующим образом:

- значение синуса угла помещается в st(0);
- косинус угла помещается в стек регистров.

Таким образом, после выполнения операции косинус угла окажется в вершине стека st(0), синус — в регистре st(1). Следующий пример демонстрирует вычисление синуса и косинуса угла, заданного в градусах, и реализован в виде процедуры _sincos_demo, которая в качестве параметра принимает угол в градусах (листинг 9.12).

**Листинг 9.12.** Вычисление синуса и косинуса угла

```
.686
.model flat
option casemap:none
.data
  coeff      DD  0.0174
  sincos_val label qword
     sin_val DD 0
     cos_val DD 0
.code
 _sincos_demo proc
   push  EBP
   mov   EBP, ESP
   finit
   fld   dword ptr [EBP+8]
   fmul  dword ptr coeff
   fsincos
   fstp  dword ptr cos_val
   fstp  dword ptr sin_val
   lea   EAX, sincos_val
   pop   EBP
   ret
 _sincos_demo endp
 end
```

В области данных процедуры определена переменная coeff, численно равная значению $\pi/180$, с помощью которой угол в градусах переводится в угол в радианах. Для извлечения параметра используется регистр EBP, так что следующая команда загружает в вершину стека значение угла в градусах:

```
fld dword ptr [EBP+8]
```

С помощью команды fmul dword ptr coeff значение угла переводится в радианы, а команда fsincos вычисляет значения синуса и косинуса угла. Следующие две команды fstp позволяют сохранить полученные значения в области памяти sincos_val. Последний шаг — сохранить адрес памяти с полученными значениями в регистре EAX (команда lea EAX, sincos_val), после чего выйти из процедуры.

Вызывающая программа для проверки работоспособности этой процедуры на Visual C++ .NET может выглядеть так, как показано в листинге 9.13.

**Листинг 9.13.** Демонстрационная программа для процедуры из листинга 9.12

```
#include <stdio.h>
extern "C" float* sincos_demo(float angle);
int main(void)
{
 float angle = 34;
 float* res = sincos_demo(angle);
 printf("Sine: %5.2f\t", *res);
 printf("Cosine: %5.2f\t\n", *++res);
 return 0;
}
```

В этом приложении процедура sincos_demo объявлена как внешняя, принимающая параметр в виде переменной с плавающей точкой angle. В программе

объявлена переменная-указатель res, принимающая адрес области памяти, содержащей результат. Далее, с помощью функции printf последовательно выводятся значения синуса и косинуса заданного угла. Для указанного угла, равного 34°, приложение выведет результат:

```
Sine: 0.56 Cosine: 0.83
```

К тригонометрическим командам относится и fptan. Она вычисляет частичный тангенс угла, значение которого находится в вершине стека сопроцессора. Результат возвращается в регистрах st(0) (значение косинуса) и st(1) (значение синуса). Следующий пример демонстрирует вычисление тангенса угла с помощью процедуры _fptan_demo. Процедура в качестве параметра принимает значение угла в градусах, а возвращает адрес переменной, содержащей значение тангенса (листинг 9.14).

**Листинг 9.14.** Вычисление тангенса угла

```
.686
.model flat
option casemap:none
.data
   coeff   DD 0.0174
   tan_val DD 0
_fptan_demo proc
   push  EBP
   mov   EBP, ESP
   finit
   fld   dword ptr [EBP+8]
   fmul  dword ptr coeff
   fptan
   fdivr st, st(1)
   fstp  dword ptr tan_val
   lea   EAX, tan_val
   pop   EBP
   ret
 _fptan_demo endp
 end
```

Как и в предыдущем примере, в этой процедуре имеется переменная coeff, содержащая коэффициент перевода угла в градусах в угол в радианах. Команда fptan вычисляет значения синуса и косинуса, причем в регистре st(0) располагается значение косинуса угла, а в регистре st(1) — значение синуса. Собственно тангенс угла вычисляется как соотношение синуса к косинусу, что и выполняет команда

```
fdivr st, st(1)
```

После выполнения этой команды регистр st(0) будет содержать тангенс угла, который сохраняется в переменной tan_val следующей командой:

```
fstp dword ptr tan_val
```

Адрес этой переменной в регистре EAX и возвращает процедура.

К этой группе относится еще одна команда, позволяющая по значению тангенса угла вычислить сам угол. Это команда fpatan. К сожалению, в некоторых литературных источниках команда fpatan описана неправильно, поэтому я подробно остановлюсь на принципе ее работы. Вычисление производится по формуле

$$Z = \mathrm{ARCTAN}(Y/X),$$

где соотношение $Y/X$ является тангенсом искомого угла, причем значение $X$ берется из регистра st(0), а $Y$ — из регистра st(1). Результатом выполнения команды является угол $Z$ (в радианах!), значение которого помещается в вершину стека st(0).

Возникает вопрос, каким образом вычислить значение угла, используя не два аргумента ($X$ и $Y$), а всего один. Выход очень простой: нужно в качестве $X$ задать 1, тогда можно использовать один аргумент ($Y$). Это продемонстрировано в следующем примере (листинг 9.15). Здесь с помощью процедуры _arctg_demo вычисляется угол, тангенс которого служит единственным параметром функции и передается через стек. Результат (значение угла в градусах) помещается в переменную res, адрес которой возвращается в вызывающую программу в регистре EAX.

**Листинг 9.15.** Вычисление арктангенса по заданному значению тангенса угла

```
.686
.model flat
option casemap:none
.data
  coeff DD 57.32              ; коэффициент для перевода значения
                              ; угла из радиан в градусы
                              ; и численно равный 180/pi
  res   DD 0
.code
 _arctg_demo proc
   push  EBP
   mov   EBP, ESP
   finit
   fld   dword ptr [EBP+8] ; значение тангенса угла
   fldl                    ; константа 1
                           ; после предыдущих двух команд регистр st(0) содержит 1,
                           ; а регистр st(1) - значение тангенса искомого угла
   fpatan                  ; вычислить угол
   fmul  dword ptr coeff   ; перевести значение угла в градусы
   fstp  dword ptr res     ; сохранить результат в переменной res
                           ; и вытолкнуть содержимое из вершины стека
   lea   EAX, res          ; адрес результата -> EAX
   pop   EBP
   ret
 _arctg_demo endp
 end
```

Программный код этой процедуры достаточно подробно описан в комментариях, поэтому дополнительные пояснения к исходному тексту, думаю, не требуются. Приведу пример тестовой программы для проверки работоспособности процедуры _arctg_demo, написанной на Visual C++ .NET (листинг 9.16).

**Листинг 9.16.** Демонстрационная программа для процедуры из листинга 9.15

```
#include <stdio.h>
extern "C" float* arctg_demo(float tg);
int main(void)
{
 float tg = 2.94;
 printf("Angle (grad) = %5.2f\n", *arctg_demo(tg));
 return 0;
}
```

В этой программе процедура arctg_demo объявлена внешней, в качестве параметра принимающей значение тангенса в формате числа с плавающей точкой. При указанном значении параметра (2,94) процедура возвращает значение угла, равное 71,25°.

Команда fpatan очень полезна при вычислении значений других обратных тригонометрических функций, таких, например, как арксинус или арккосинус. Для подобных вычислений используются стандартные математические соотношения. Например, для вычисления арксинуса (arcsin) можно воспользоваться формулой

$$\arcsin X = \operatorname{arctg}\frac{1}{\sqrt{1-X^2}}.$$

Рассмотрим пример вычисления арксинуса. Из формулы видно, что одним из промежуточных действий является вычисление квадратного корня выражения $1 - X^2$. В набор команд математического сопроцессора входит команда fsqrt, позволяющая вычислить значение квадратного корня из числа, находящегося в вершине стека st(0). Команда не имеет аргументов и возвращает значение в вершине стека.

Вычисление арксинуса применительно к командам сопроцессора можно выполнить, используя мнемоническое выражение

$$Y = \text{fpatan}\ (X/\text{fsqrt}(1 - X^2)),$$

где $X$ — синус угла, $Y$ — значение угла, который следует найти по известному значению $X$.

Далее представлен исходный текст процедуры (назовем ее _arcsin_demo), вычисляющей значение arcsin $X$ по заданному значению синуса. Исходный текст процедуры достаточно сложен, поэтому я объясню смысл команд подробно. Как и в предыдущем примере, процедура сохраняет результат (угол в градусах) в переменной res, возвращая в вызывающую программу адрес этой переменной в регистре EAX.

В качестве единственного параметра процедура _arcsin_demo принимает значение синуса угла (не угла!). Для извлечения параметра, как обычно, используется регистр EBP. Проанализируем исходный текст процедуры (листинг 9.17).

Вначале посмотрим, что находится в области данных. Здесь определен коэффициент coeff, который нужен для перевода значения угла из радиан в градусы. Кроме того, присутствует и вспомогательная переменная one, которая включена в программный код для большей ясности. Обратите внимание на то, что переменная one должна быть представлена в форме вещественного числа (1,0)!

**Листинг 9.17.** Вычисление арксинуса через арктангенс

```
.686
.model flat
option casemap:none
.data
  one   DD 1.0
  coeff DD 57.32
  res   DD 0
.code
_arcsin_demo proc
   push  EBP
   mov   EBP, ESP
   finit
   fld   dword ptr [EBP+8]
   fld   dword ptr [EBP+8]
   fmul
   fchs
   fadd  dword ptr one
   fsqrt
   fld   dword ptr [EBP+8]
   fdiv  st, st(1)
   fld1
   fpatan
   fmul  dword ptr coeff
 exit:
   fstp  dword ptr res
   :
   lea   EAX, res
   pop   EBP
   ret
 _arcsin_demo endp
 end
```

Вычисления выполняются в последовательности, которую легко проследить, используя известную нам формулу $Y$ = fpatan ($X$/fsqrt(1 – $X^2$)). Напомню, что число $X$ соответствует параметру, передаваемому в процедуру.

Вначале находим значение $X^2$ с помощью команд

```
fld   dword ptr [EBP+8]
fld   dword ptr [EBP+8]
fmul
```

Далее, знак $X^2$ меняется на минус с помощью команды fchs. Данная команда меняет знак значения, находящегося в вершине стека st(0). После выполнения этой команды в вершине стека будет находиться значение $-X^2$.

К полученному значению прибавляем 1 с помощью команды

```
fadd dword ptr one
```

Таким образом, в вершине стека к этому моменту содержится значение $1 - X^2$. На следующем шаге вычисляем квадратный корень этого значения, используя команду fsqrt. Следуя логике программы, нужно вычислить значение выражения $X$/fsqrt(1 – $X^2$):

```
fld   dword ptr [EBP+8]
fdiv  st, st(1)
```

Теперь, как и в предыдущем примере, для формирования аргумента команды fpatan поместим в вершину стека 1, выполнив команду fld1. Наконец, можно применить команду fpatan и перевести полученное значение угла из радиан в градусы:

```
fpatan
fmul  dword ptr coeff
```

Полученное значение угла сохраняется в переменной res, адрес результата — в регистре EAX, и процедура завершается. Для проверки работоспособности процедуры можно воспользоваться, например, короткой программой на Visual C++ .NET (листинг 9.18).

**Листинг 9.18.** Демонстрационная программа для процедуры из листинга 9.17

```
#include <stdio.h>
extern "C" float* arcsin_demo(float sinx);
int main(void)
{
 float sinx = -0.37;
 printf("Angle (grad) = %5.2f\n", *arcsin_demo(sinx));
 return 0;
}
```

Перед использованием внешней процедуры следует объявить ее с директивой extern. В качестве параметра в вызываемую процедуру передается значение вещественной переменной sinx. В остальном исходный текст программы прост и в объяснениях не нуждается.

При указанном значении синуса (–0,37) соответствующий угол равен –21,72°. В качестве упражнения читатели могут попробовать разработать процедуру для вычисления арккосинуса.

Последняя подгруппа команд, которую мы проанализируем, позволяет вычислять значения логарифмических и показательных функций. К этим командам относятся:

- f2xm1 — вычисляет значения функции $y = 2x^{-1}$. Исходное значение параметра $x$ должно находиться в вершине стека st(0) и лежать в диапазоне $-1 \leq x \leq 1$, результат размещается в вершине стека (регистр st(0)). Команда может использоваться для вычисления различных показательных функций;
- fyl2x — вычисляет значение функции $z = y\log_2(x)$. Исходное значение $x$ размещается в вершине стека сопроцессора, а исходное значение $y$ — в регистре st(1). Значение $x$ должно находиться в диапазоне $0 \leq x \leq +\infty$, а значение $y$ — в диапазоне $-\infty \leq y \leq +\infty$. Перед записью результата в вершину стека команда fyl2x выталкивает из стека значения $x$ и $y$ и только после этого помещает результат $z$ в st(0);
- fyl2xp1 — вычисляет значение функции $z = y\log_2(x + 1)$, при этом исходное значение $x$ размещается в вершине стека st(0), а исходное значение $y$ — в регистре st(1). Значение $x$ должно находиться в диапазоне $0 \leq |x| \leq 1 - 1/\sqrt{2}$, а значение $y$ — в диапазоне $-\infty \leq y \leq +\infty$. Перед записью команда выталкивает значения $x$ и $y$ из стека, после чего результат $z$ записывается в st(0).

Перечисленные команды помогают вычислять самые разнообразные показательные и логарифмические функции. Для иллюстрации рассмотрим пример вычисления натурального логарифма числа. Вспомним формулу для вычисления натурального логарифма числа, если известен двоичный логарифм числа:

$$\ln X = \log_2 X / \log_2 e,$$

где $X$ — заданное число.

Эту формулу можно представить в несколько ином виде:

$$\ln X = 1/\log_2 e \times \log_2 X.$$

Очевидно, что для вычисления натурального логарифма числа можно использовать команду fyl2x, при этом $y$ следует взять равным $1/\log_2 e$. В листинге 9.19 приведен исходный текст процедуры (она называется _lnx_demo), вычисляющей натуральный логарифм числа, которое является параметром процедуры.

**Листинг 9.19.** Вычисление натурального логарифма числа

```
.686
.model flat
option casemap: none
.data
  res DD 0
.code
 _lnx_demo proc
   push  EBP
   mov   EBP, ESP
   fldln2
   fld1
   fdiv
   fld   dword ptr [EBP+8]
   fyl2x
   fstp  dword ptr res
   lea   EAX, res
   pop   EBP
   ret
 _lnx_demo endp
 end
```

Параметр извлекается из стека посредством регистра EBP. Вычисление коэффициента $1/\log_2 e$ выполняется с помощью команд

```
fldln2
fld1
fdiv
```

После выполнения этих команд в стек загружается значение параметра и с помощью команды fyl2x вычисляется натуральный логарифм числа. Полученный результат сохраняется в переменной res, а адрес переменной помещается в регистр EAX для возврата в вызывающую процедуру. Проверить работу процедуры можно с помощью простой программы на Visual C++ .NET (листинг 9.20).

При указанном значении параметра $x$ (432,804) натуральный логарифм числа равен 6,07.

**Листинг 9.20.** Демонстрационная программа для процедуры из листинга 9.19

```
#include <stdio.h>
extern "C" float* lnx_demo(float x);
int main(void)
{
 float x = 432.804;
 printf("Ln %6.3f = %5.2f\n", x, *lnx_demo(x));
 return 0;
}
```

К группе дополнительных арифметических команд относят несколько инструкций, выполняющих специфичные действия над одним операндом, находящимся в вершине стека st(0). Некоторые команды (fsqrt, fchs) нам уже встречались, другие мы рассмотрим впервые. Вот перечень команд этой группы и их описание:

- fsqrt — вычисляет квадратный корень из значения, находящегося в вершине стека st(0), помещая туда же результат;
- fchs — изменяет знак числа, находящегося в вершине стека st(0), помещая туда же результат;
- fabs — вычисляет абсолютное значение (модуль) числа, находящегося в вершине стека st(0), помещая туда же результат;
- fscale — выполняет умножение содержимого вершины стека st(0) на степень 2.

Вычисление производится по формуле $Y = Y \times 2X$, где $Y$ — значение, находящееся в st(0), а $X$ — масштабирующий множитель, находящийся в регистре st(1). Результирующее значение замещает содержимое регистра st(0), при этом масштабирующий множитель остается в st(1) без изменения. Если масштабирующий множитель не является целым числом, то его значение округляется в меньшую сторону до ближайшего целочисленного значения. Приведу короткий пример программного кода, в котором используется команда fscale (листинг 9.21). Процедура _fscal_ex принимает два параметра: масштабирующий множитель (EBP+12) и само число (EBP+8). Результирующее значение сохраняется в переменной res, адрес которой процедура возвращает в регистре EAX.

**Листинг 9.21.** Применение команды fscale

```
.686
.model flat
option casemap: none
.data
  res  DD 0
.code
_fscale_ex proc
  push  EBP
  mov   EBP, ESP
  fld   dword ptr [EBP+12] ; масштабирующий множитель -> st(1)
  fld   dword ptr [EBP+8]  ; число -> st(0)
  fscale
  fstp  dword ptr res
  lea   EAX, res
  pop   EBP
  ret
_fscale_ex endp
end
```

Исходный текст процедуры понятен, и мы не будем на нем останавливаться. Для проверки результатов выполнения процедуры воспользуемся программой на Visual C++ .NET (листинг 9.22).

**Листинг 9.22.** Демонстрационная программа для процедуры из листинга 9.21

```
#include <stdio.h>
extern "C" float* fscale_ex(float num, float scale);
int main(void)
{
 float num = 95.31;
 float scale = -3.17;
 printf("Scaling value: %6.3f\n", *fscale_ex(num, scale));
 return 0;
}
```

При указанных значениях параметров и принятой точности (3 значащие цифры после точки) программа отображает такой реэультат:

```
Scaling value: 11.914
```

Следующая команда, которую мы рассмотрим, — fxtract. Она выделяет из числа, находящегося в вершине стека st(0), порядок (exponent) и мантиссу (mantissa, significand). При этом значение порядка помещается в регистр st(1), а мантисса — в регистр st(0). В листинге 9.23 приведен пример процедуры _fxtract_ex, принимающей в качестве параметра число с плавающей точкой и возвращающей в регистре EAX адрес области памяти res, в которой сохраняются значения мантиссы и порядка.

**Листинг 9.23.** Применение команды fxtract

```
.686
.model flat
option casemap: none
.data
  res         label qword
  significand DD 0
  exponent    DD 0
.code
_fxtract_ex proc
  push  EBP
  mov   EBP, ESP
  fld   dword ptr [EBP+8]        ; число -> st(0)
  fxtract
  fstp  dword ptr significand  ; st(0) -> significand (mantissa)
  fstp  dword ptr exponent     ; st(1) -> exponent
  lea   EAX, qword ptr res
  pop   EBP
  ret
_fxtract_ex endp
end
```

Проверить работоспособность процедуры _fxtract_ex можно с помощью простой программы на Visual C++ NET, вызывающей эту процедуру (листинг 9.24).

**Листинг 9.24.** Демонстрационная программа для процедуры из листинга 9.23

```
#include <stdio.h>
extern "C" float* fxtract_ex(float num);
int main(void)
{
 float num = 95.31;
 float* pf = fxtract_ex(num);
 printf("Significand: %8.5f  ", *pf++);
 printf("Exponent: %5.2f\n", *pf);
 return 0;
}
```

При указанном значении параметра num (95,31) и указанной точности программа выводит на экран результирующие значения мантиссы и порядка:

```
Significand: 1.48922 Exponent: 6.00
```

Рассмотрим назначение и синтаксис последней команды из этой группы — frndint. Эта команда выполняет округление числа, находящегося в вершине стека st(0), до целого числа. Команда не имеет операндов и возвращает результат в регистре st(0). Допускается четыре режима округления, причем определяются они значением поля **rc** управляющего регистра cwr сопроцессора (см. рис. 9.8). Режим округления можно устанавливать в процессе выполнения программы до того, как потребуется эта операция. Для этого можно установить нужную комбинацию битов в поле **rc** регистра cwr, используя команды fstcw (сохранение регистра cwr) и fldcw (загрузка регистра cwr).

Следующий пример демонстрирует работу команды frndint. В примере показан исходный текст процедуры, выполняющий округление суммы двух вещественных чисел в зависимости от заданного режима (процедура называется _frndint_ex). Процедура принимает три параметра: значения двух вещественных чисел и целое число, величина которого указывает режим округления.

Параметр, указывающий режим округления, может принимать значения 0–3, при этом 0 соответствует режиму с **rc** = 00 (округления к ближайшему целому числу), 1 — режиму с **rc** = 01 (округление в меньшую сторону), 2 — режиму с **rc** = 10 (округление в большую сторону) и, наконец, значение 3 определяет режим с **rc** = 11 (отбрасывания дробной части).

Мнемонически процедуру можно записать так:

```
_frndint_ex(float a, float b, int mode)
```

Процедура возвращает в регистре EAX адрес переменной res, содержащей результат. Исходный текст самой процедуры представлен в листинге 9.25.

**Листинг 9.25.** Использование различных режимов округления

```
.686
.model flat
option casemap: none
.data
 mask_rc   label word
     DW   0F3FFh  ; rc = mode 00
```

продолжение

**Листинг 9.25** (продолжение)

```
        DW   0F7FFh  ; rc = mode 01
        DW   0FBFFh  ; rc = mode 10
        DW   0FFFFh  ; rc = mode 11
    tmp  DW 0
    res  DD 0
   .code
    _frndint_ex proc
      push  EBP
      mov   EBP,ESP
      mov   ECX, dword ptr [EBP+16]
      shl   ECX, 1
      lea   ESI, mask_rc
      add   ESI, ECX
      mov   DX, word ptr [ESI]
      finit
      fstcw tmp
      or    tmp, 0C00h
      and   tmp, DX
      fldcw tmp
      fld   dword ptr [EBP+12]
      fadd  dword ptr [EBP+8]
      frndint
      fstp  dword ptr res
      lea   EAX, res
      pop   EBP
      ret
    _frndint_ex endp
    end
```

Проанализируем программный код _frndint_ex и начнем с области данных. Здесь определено поле mask_rc, каждый элемент которого является словом и содержит маску для установки одного из четырех режимов округления. Фактически слово маски устанавливает одну из четырех комбинаций битов 10–11 поля **rc** в регистре управления cwr. Для доступа к соответствующему слову маски в области памяти mask_rc используется группа команд:

```
mov   ECX, dword ptr [EBP+16]
shl   ECX, 1
lea   ESI, mask_rc
add   ESI, ECX
mov   DX, word ptr [ESI]
```

Смысл этих команд следующий: вначале в регистр ECX помещается параметр, указывающий номер режима (число в диапазоне 0–3). Содержимое регистра ECX служит индексом для выбора нужного элемента из поля mask_rc, но поскольку элементы этого поля имеют размерность слова, то значение индекса в ECX нужно умножить на 2. Затем в регистр ESI помещается значение mask_rc, которое является базовым для доступа к элементам этого поля данных (команда lea).

Для вычисления смещения слова маски используется команда add, прибавляющая смещение в ECX к базовому адресу в регистре ESI. Наконец, последняя команда

этой группы помещает маску режима в регистр DX. В дальнейшем содержимое регистра DX понадобится для управления регистром cwr.

Следующие несколько команд обеспечивают непосредственную установку режима округления. В регистр cwr нельзя записать непосредственное значение, но можно считать содержимое регистра в ячейку памяти, после чего модифицировать полученное значение и записать его обратно в регистр управления округлением. Команда fstcw tmp сохраняет содержимое регистра cwr в переменной tmp размером в слово. Следующие две команды корректируют считанное содержимое регистра cwr для работы в выбранном режиме:

```
or   tmp. 0C00h
and  tmp. DX
```

Затем в регистр cwr записывается новое значение переменной tmp с помощью команды

```
fldcw tmp
```

С этого момента начинают действовать новые установки.

Остальная часть программного кода процедуры демонстрирует собственно работу самой команды frndint. Вначале два числа складываются, при этом результат помещается в регистр st(0). Для этого служат команды

```
fld  dword ptr [EBP+12]
fadd dword ptr [EBP+8]
```

Команда frndint округляет число в регистре st(0) в соответствии с установками поля **rc** регистра cwr, которые только что были изменены. Для сохранения результата используется команда

```
fstp dword ptr res
```

Эта команда копирует значение st(0) в переменную res.

Проверить работоспособность процедуры можно с помощью простой программы на Visual C++ .NET (листинг 9.26).

**Листинг 9.26.** Демонстрационная программа для процедуры из листинга 9.25

```
#include <stdio.h>
extern "C" float* frndint_ex(float a1. float b1. int mode):
int main(void)
{
 float a1 = 3.18:
 float b1 = 5.43:
 printf("For a1 = %5.2f. b1 = %5.2f. a1+b1 = %5.2f:\n". a1. b1. a1+b1):
 for (int mode = 0: mode < 4: mode++)
 {
 printf("FRNDINT: rc=%d. rounded sum = %5.2f\n".
   mode. *frndint_ex(a1. b1. mode)):
 }
 return 0:
}
```

Здесь параметры a1и b1 определены как числа с плавающей точкой (float), а параметр mode принимает целочисленные значения в диапазоне 0–3. В зависимости

от значения переменной mode программа будет выводить на экран разные значения:

```
For a1 = 3.18, b1 = 5.43, a1+b1 = 8.61:
FRNDINT: rc=0, rounded sum = 9.00
FRNDINT: rc=1, rounded sum = 8.00
FRNDINT: rc=2, rounded sum = 9.00
FRNDINT: rc=3, rounded sum = 8.00
```

Последняя группа команд, которую мы рассмотрим, — управляющие команды математического сопроцессора.

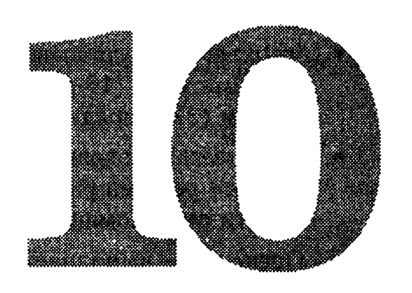
Как правило, эти команды не используются в вычислениях, а управляют действиями сопроцессора на системном уровне. Подобные действия включают в себя инициализацию модуля обработки операций с плавающей точкой, обработку численных исключений и переключение задач. Ознакомимся с синтаксисом основных управляющих команд:

- fwait — синхронизация работы процессора и математического сопроцессора (если процессор встречает эту команду, он приостанавливает свою работу до окончания выполнения очередной команды сопроцессора);
- finit — инициализация сопроцессора с помещением предопределенных значений в управляющие регистры сопроцессора (эта команда подробно рассматривалась в начале этой главы);
- fstsw dest — сохранение регистра состояния в 16-разрядном операнде dest, слово состояния может сохраняться и в 16-разрядном регистре AX;
- fstcw dest — сохранение содержимого регистра управления в 16-разрядной переменной dest в памяти (команда обычно используется для анализа полей регистра cwr);
- fldcw src — загрузка содержимого 16-разрядной переменной src в регистр cwr (команда используется для задания режимов работы сопроцессора);
- fclex — сброс флагов исключений в регистре swr сопроцессора;
- fincstp — увеличивает указатель стека на единицу в поле **top** регистра состояния сопроцессора (swr). Команда не имеет операндов и по своему действию напоминает команду fst. Содержимое вершины стека при выполнении команды удаляется;
- fdecstp — уменьшает указатель стека на единицу в поле **top** регистра состояния сопроцессора (swr). Команда не имеет операндов и по своему действию напоминает команду fld, но операнд в стек не помещается;
- ffree st(*i*) — освобождает регистр стека st(*i*), помечая его как пустой в регистре тегов (twr). Содержимое поля **top** в регистре состояния (swr) и само содержание регистра не изменяются;
- fnop — команда ничего не делает и может использоваться для временных задержек;
- fsave dest — запоминает состояние сопроцессора в 94-байтовой области памяти dest (16-разрядный режим работы) или в 108-байтовой области памяти (32-разрядный режим работы);

- `frstor src` — восстанавливает состояние сопроцессора из 94-байтовой области памяти `src` (16-разрядный режим работы) или из 108-байтовой области памяти (32-разрядный режим работы);
- `fstenv dest` — сохраняет состояние среды сопроцессора (14 байт для 16-разрядного режима работы и 28 байт для 32-разрядного режима) в области памяти `dest`. Команда не сохраняет содержимое стека регистров (80 байт);
- `fldenv src` — выполняет частичное восстановление состояния среды сопроцессора (14 байт для 16-разрядного режима работы и 28 байт для 32-разрядного режима) из области памяти `src`.

Все эти команды обычно требуются при разработке программ с обработчиками исключительных ситуаций, а также в многозадачной среде, хотя могут использоваться и в обычных приложениях для создания оригинальных алгоритмов обработки чисел.

# Интерфейс с языками высокого уровня



В процессе разработки программ на языках высокого уровня одной из важнейших проблем, с которой сталкивается разработчик, является производительность приложения. Эффективным средством ее повышения является применение языка ассемблера для разработки критических участков программного кода и оформление их в виде подпрограмм. В этой главе рассматриваются наиболее важные аспекты создания интерфейсов подпрограмм на ассемблере и приложений, разработанных на популярных языках высокого уровня C++ и Pascal. Необходимость разработки отдельной подпрограммы на ассемблере возникает, когда требуется:

- реализовать какой-то специальный алгоритм, который требует нетривиальной обработки данных и который трудно создать средствами языка высокого уровня;
- обеспечить высокое быстродействие какого-либо алгоритма обработки данных или фрагмента программы.

## 10.1. Общие принципы построения интерфейсов

Принципы создания интерфейсов ассемблерных подпрограмм с языками высокого уровня будем рассматривать применительно к двум наиболее популярным инструментам быстрой разработки: Microsoft Visual C++ .NET 2003 и Borland Delphi 2005. В этих пакетах программ в качестве базовых языков программирования используются языки C++ (Visual C++ .NET) и Pascal (Delphi 2005), поэтому в дальнейшем при ссылках на языки C++ и Pascal будем иметь в виду эти инструменты разработки.

Несколько слов о терминологии. Напомню, что в этой главе, как и во всех остальных, мы используем термины «подпрограмма» и «процедура» как синонимы.

Для компиляции ассемблерных подпрограмм и процедур можно задействовать, как и везде в этой книге, макроассемблер MASM версии 6.14*xxxx* или 7.10*xxxx*. Хочу сделать важное замечание: при разработке и анализе программного кода мы будем рассматривать только 32-разрядные подпрограммы на ассемблере, то есть разработанные с помощью модели памяти flat.

Для написания ассемблерных модулей будет использоваться упрощенный синтаксис ассемблера MASM. Это означает, что везде в исходных текстах для инициализации логического сегмента данных будет указываться директива .data, а для инициализации логического сегмента кода — директива .code. Что же касается области стека, то мы будем работать на стеке вызывающей программы, и отдельная инициализация логического сегмента стека не понадобится.

При использовании линейной (flat) модели памяти ближняя (near) и дальняя (far) адресация команд и данных не различается, при этом все ссылки в 4-гигабайтном адресном пространстве считаются ближними. Это означает, что в процедурах можно не указывать директивы near и far, поскольку компилятор интерпретирует все ссылки как ближние (near).

В примерах этой главы мы будем использовать отдельно скомпилированные модули на ассемблере, которые компонуются с программами на C++ .NET и Delphi 2005. Эти файлы имеют расширение OBJ и называются объектными файлами или объектными модулями. Если подпрограмма будет применяться совместно с приложением на Visual C++ .NET, то командная строка для компилятора MASM выглядит так:

```
ml /c /Fo имя_файла.obj имя_файла.asm
```

Если подпрограмма на ассемблере будет применяться в приложении, написанном на Delphi 2005, то командная строка должна выглядеть так:

- компилятор ml версии 6.14*xxxx*:

  ```
  ml /c /Fo имя_файла.obj имя_файла.asm
  ```

- компилятор ml версии 7.10*xxxx*:

  ```
  ml /c /omf /Fo имя_файла.obj имя_файла.asm
  ```

Различия в параметрах связаны с тем, что Visual C++ .NET работает с объектными файлами в формате COFF (Common Object File Format), а Delphi использует файлы в стандарте OMF (Object Module Format). Компилятор ассемблера версии 6.14 по умолчанию создает объектный файл в формате OMF, в то время как компилятор версии 7.10 — в формате COFF.

Если во время сборки приложения на Delphi появятся сообщения наподобие следующих, то это свидетельствует о некорректном формате объектного файла:

```
[Error] Project1.dpr(15): E2045 Bad object file format: \...\sub1.obj'
[Error] Project1.dpr(11): E2065 Unsatisfied forward or external declaration: 'sub1'
```

Второе сообщение является следствием обнаруженной компилятором некорректности.

Чтобы избежать подобных ошибок во время сборки программ в Delphi, следует указывать параметр /omf (версия 7.10 MASM) при компиляции ассемблерной

процедуры. Во время компиляции ассемблерных модулей можно использовать упрощенный вариант командной строки, например:

- версия 6.14*.xxxx*:

```
ml /c имя_файла.asm
```

- версия 7.10*.xxxx*:

```
ml /c /omf имя_файла.asm
```

В этом случае имя объектного файла будет совпадать с именем файла исходного текста. В процессе сборки проекта в Visual C++ .NET вы можете получить предупреждение компоновщика:

```
Warning: converting object format from OMF to COFF
```

Это предупреждение свидетельствует о том, что OMF-файл будет преобразован в формат COFF, и принципиально оно ничего не меняет, поскольку компилятор Visual C++ .NET преобразует OMF-файл в формат COFF в любом случае.

Перед сборкой приложения в Visual C++ .NET необходимо добавить в проект объектный файл с вызываемой процедурой. Лучше всего поместить объектный файл с процедурой в рабочий каталог проекта.

При сборке проекта в Delphi 2005 следует указать местоположение объектного файла при помощи директивы

```
{$L путь_к_обьектному_файлу}
```

Более подробно методику включения ассемблерных модулей в проекты мы рассмотрим далее в этой главе при анализе примеров.

Перед разработкой ассемблерных процедур для использования в программах на C++ и Pascal остановимся подробно на требованиях, которые должен соблюдать разработчик при создании интерфейса. Их несколько:

- имена идентификаторов (переменных и процедур), включенных в объектные файлы, должны соответствовать правилам построения имен данным компилятором языка высокого уровня;
- модель памяти, используемая ассемблерной подпрограммой, должна быть линейной (мы рассматриваем только 32-разрядные приложения);
- способ передачи параметров процедуре должен быть указан в вызывающей программе;
- если вызывающая программа ожидает результат выполнения процедуры, то его тип должен быть указан в программе и реально соответствовать тому, который возвращает ассемблерная процедура.

Рассмотрим требования к интерфейсу более детально и начнем с имен идентификаторов. Для языка Pascal все строчные буквы в именах внешних идентификаторов преобразуются в прописные. Компилятор C++ не изменяет регистр букв, поэтому имена идентификаторов чувствительны к регистру, что следует учитывать при разработке процедур. Кроме того, компилятор C++ помещает в имя процедуры префикс и суффикс в виде определенной последовательности символов. Принципы формирования имен в C++ мы рассмотрим далее в этой главе.

Способ передачи параметров ассемблерной процедуре должен учитывать следующее:

- параметры в вызываемую подпрограмму передаются либо по значению, либо по ссылке. При передаче по значению передается 32-разрядное значение операнда, а при передаче по ссылке — его адрес (32-разрядный);
- параметры в процедуру могут передаваться через стек, регистры или общую область памяти. Мы будем рассматривать только наиболее распространенные способы передачи параметров — через стек и регистры.

Проанализируем более подробно способы передачи параметров через стек и регистры процессора. Все они сведены в табл. 10.1.

**Таблица 10.1.** Способы передачи параметров

| Директива | Передача параметров | Очистка стека | Использование регистров |
|---|---|---|---|
| register (fastcall) | Слева направо | Процедура | EAX, EDX, ECX (Delphi) |
| | | | ECX, EDX (Visual C++ .NET) |
| pascal | Слева направо | Процедура | Нет |
| cdecl | Справа налево | Вызывающая программа | Нет |
| stdcall | Справа налево | Процедура | Нет |
| safecall | Справа налево | Процедура | Нет |

Директивы, указанные в первой колонке слева, называют соглашениями о вызовах (calling conventions), соглашениями об именовании или соглашениями о передаче параметров. Все эти определения являются синонимами и определяют способы взаимодействия программных модулей (в общем случае, разработанных с помощью разных языков программирования). Любая из этих директив определяет:

- способ передачи параметров в процедуры;
- способ синхронизации области стека при его использовании процедурой;
- способ формирования имен и данных для взаимодействия нескольких программ и/или процедур.

Остальные колонки подробно описывают каждый тип соглашения о вызовах.

Вторая колонка слева показывает порядок размещения параметров в стеке или в регистрах (при передаче параметров через регистры). Например, пусть мнемоника вызываемой процедуры записывается так:

```
myproc (param1, param2, param3)
```

Тогда при способе передачи параметров stdcall первым помещается в стек параметр param3, вторым — параметр param2 и третьим — параметр param1. Если, например, используется способ передачи параметров register, то параметр param1 помещается в регистр EAX, параметр param2 — в регистр EDX и, наконец, параметр param3 — в ECX. Более наглядно методику передачи параметров демонстрирует рис. 10.1 (для соглашения stdcall).

Этот рисунок демонстрирует еще один очень важный аспект: параметры в стеке размещаются, начиная с адреса ESP+4, поскольку указатель стека ESP содержит эффективный адрес (ЕА) команды следующей после вызова call myproc. Это действительно для любых способов передачи параметров, использующих стек (pascal, cdecl, safecall), параметры будут находиться в стеке, начиная с адреса ESP+4.

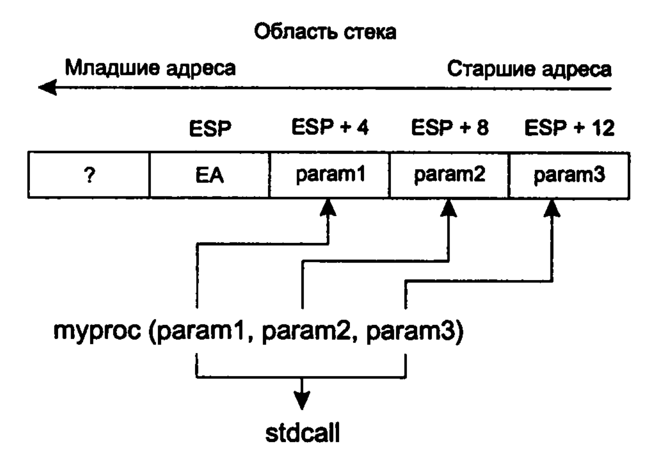



**Рис. 10.1.** Вызов процедуры myproc в соответствии с соглашением stdcall

Перед возвращением в основную программу необходимо восстанавливать или, как иногда говорят, очищать стек. Речь идет о том, что после завершения процедуры указатель стека смещается на 4 в сторону увеличения адресов и будет указывать на ставшие ненужными параметры. Например, после завершения процедуры myproc регистр ESP будет указывать на параметр param1 (рис. 10.2).

Область стека

Младшие адреса ← Старшие адреса

| | | ESP | ESP + 4 | ESP + 8 |
|---|---|---|---|---|
| ? | ? | param1 | param2 | param3 |

**Рис. 10.2.** Содержимое стека после завершения процедуры myproc

Если каждый раз после вызова процедур оставлять стек в таком состоянии, то область памяти стека быстро переполнится, что вызовет ошибку и останов программы. По этой причине указатель стека нужно очищать. Это можно сделать, сместив указатель стека вверх (в сторону увеличения адресов) на величину, определяемую количеством занимаемых ненужными параметрами байтов.

В нашем примере нужно сместить указатель стека на 12 вверх (3 параметра × × 4 байта). Такую операцию может выполнить как вызывающая программа, так и сама процедура. Способ очистки стека определяется соглашением о передаче параметров (третья колонка слева в табл. 10.1). В случае соглашения stdcall

стек очищает сама вызываемая процедура, для чего в исходный текст включается команда ret *n*, где *n* — число возвращаемых байтов памяти. Эта команда должна быть последней командой процедуры. Для процедуры myproc команда возврата должна выглядеть как ret 12, хотя вместо нее можно использовать комбинацию команд:

```
add ESP, 12
ret
```

Должен заметить, что вышеуказанные команды не изменяют содержимое ячеек памяти, выделенных под область стека, поскольку это не имеет смысла (при следующих обращениях к стеку данные перезаписываются).

Выбор того или иного способа передачи параметров определяется практическими аспектами. Если, например, в программе применяются функции Windows API, то стандартным соглашением вызова для них является stdcall. Директива cdecl, например, является стандартной для компиляторов C++ и используется ими по умолчанию при вызове процедур из других модулей.

Наиболее быстрым способом передачи параметров является регистровый (register). В Visual C++ .NET этот способ имеет другое название — fastcall. Если количество передаваемых в процедуру параметров не превышает трех, то стек не используется, что и дает выигрыш в скорости.

Способ передачи параметров pascal используется в настоящее время редко и поддерживается компиляторами в целях обратной совместимости (backward compatibility), поэтому останавливаться на нем я не буду.

Вызываемая процедура в большинстве случаев возвращает основной программе результат стандартным способом — в регистре EAX. Результатом может быть либо непосредственное значение, либо адрес. Во втором случае говорят, что процедура возвращает ссылку.

На этом рассмотрение теоретических аспектов построения интерфейсов с языками высокого уровня можно закончить и перейти к демонстрации примеров взаимодействия процедур на ассемблере с программами на Visual C++ .NET и Delphi 2005.

## 10.2. Интерфейс ассемблерных процедур с Delphi 2005

Для демонстрации интерфейса ассемблерных процедур с программами на Delphi (и Visual C++ .NET) будем использовать простейшие 32-разрядные консольные приложения, вызывающие процедуру и отображающие результат ее работы на экране дисплея.

В первом примере вызываемая процедура (она называется example1) вычисляет разность двух целых чисел, передаваемых ей в качестве параметров, и возвращает результат в регистре EAX в основную программу (листинг 10.1).

**Листинг 10.1.** Вычисление разности двух целых чисел

```
.686
.model flat
option casemap:none
.data
  res DD 0
.code
 example1 proc
  push EBP
  mov  EBP, ESP
  finit
  fild  dword ptr [EBP+8]
  fisub dword ptr [EBP+12]
  fistp dword ptr res
  mov   EAX, res
  pop   EBP
  ret   8
 example1 endp
 end
```

Здесь процедуре через стек передаются два параметра, представляющие собой целые числа. Напомню, что параметры процедуры можно извлечь из стека с помощью регистра EBP. Для этого подготавливаем регистр EBP:

```
push EBP
mov  EBP, ESP
```

Параметры, передаваемые процедуре example1, к этому моменту будут расположены в стеке так, как показано на рис. 10.3.

**Область стека**

Младшие адреса ← Старшие адреса

| ESP | ESP + 4 | ESP + 8 | ESP + 12 |
|---|---|---|---|
| EA | EBP | Параметр 1 | Параметр 2 |

**Рис. 10.3.** Передача параметров в процедуру example1

Еще один вопрос, который предстоит решить, — какой способ передачи параметров должна выбрать вызывающая программа на Delphi. В нашем примере выберем способ передачи параметров в соответствии с соглашением stdcall. В этом случае по адресу EBP+12 будет находиться правый параметр, а по адресу EBP+8 — левый (см. рис. 10.3). Далее при помощи следующих команд определяется разность двух целых чисел:

```
fild  dword ptr [EBP+8]
fisub dword ptr [EBP+12]
```

Результат вычитания, находящийся в вершине стека сопроцессора st(0), помещается в переменную res командой

```
fistp dword ptr res
```

Наконец, содержимое res сохраняется в регистре EAX, и после коррекции стека (команда pop EBP) происходит выход из процедуры. Поскольку мы приняли соглашение stdcall, то вызываемая процедура сама должна восстановить указатель стека, что и выполняет команда ret 8.

Как вы заметили, нигде в исходном тексте процедуры example1 нет упоминания о соглашении stdcall. Следует учитывать, что вызывающая программа не проверяет, какое соглашение использует процедура: параметры передаются в соответствии с директивами, указанными в основной программе. Точно так же вызывающая программа не проверяет, был ли очищен указатель стека после выхода из вызываемой процедуры. По этой причине разработчик должен сам заботиться о корректном написании программного кода процедуры.

Проанализируем теперь программный код вызывающей программы, написанной на Delphi. Исходный текст программы приведен в листинге 10.2.

**Листинг 10.2.** Вызывающая программа для процедуры example1 из листинга 10.1

```
program Project1;

{$APPTYPE CONSOLE}

uses
  SysUtils;

{$L f:\example1.obj}

function example1(i1:Integer;i2:Integer):Integer;stdcall;external;

var
 a1, a2: Integer;
 ires: Integer;

begin
  a1:= -1601;
  a2:= -8892;
  ires:= example1(a1, a2);
  WriteLn(IntToStr(ires));
end.
```

В этом листинге хочу обратить внимание на несколько ключевых моментов. Прежде всего, следует указать компоновщику путь к объектному модулю, в котором находится процедура example1. Эти действия выполняет директива

```
{$L f:\example1.obj}
```

Хочу также заметить, что имя процедуры и имя объектного файла выбраны совпадающими для удобства, в общем случае они могут различаться.

Далее, нужно объявить вызываемую процедуру как внешнюю (external) и использующую соглашение о вызовах stdcall. Кроме того, следует указать параметры и их тип, а также тип возвращаемого внешней процедурой значения. Эти действия выполняет директива

```
function example1(i1:Integer;i2:Integer):Integer;stdcall;external;
```

Здесь указано, что процедура возвращает результат (ключевое слово function), что она принимает два целочисленных параметра, i1 и i2, а также то, что возвращаемое значение является целым числом.

В соответствии с этим объявлением процедуры в вызывающей программе, процедура example1 извлекает параметр i1 по адресу EBP+8, а параметр i2 — по адресу EBP+12 (см. листинг 10.1). Результат выполнения процедуры example1 численно равен i1 − i2.

Остальная часть программы на Delphi обеспечивает инициализацию переменных и вызов процедуры example2:

```
a1:= -1601:
a2:= -8892:
ires:= example1(a1. a2):
```

После выполнения этих операторов целочисленная переменная ires будет содержать значение 7291, которое преобразуется к строковому типу функцией IntToStr(ires) и выводится на экран посредством оператора WriteLn.

В этом примере при вызове процедуры example1 было использовано соглашение stdcall. Посмотрим, как изменятся вызывающая программа и процедура example1, если в качестве соглашения об именовании принять cdecl. Что касается процедуры example1, то здесь произойдет только одно изменение — команда возврата из процедуры ret будет использоваться без параметров; этот фрагмент кода выглядит так:

```
. . .
.data
  res DD 0
.code
 example1 proc
. . .
  ret
 example1 endp
 end
```

В вызывающей программе на Delphi в строке объявления процедуры example1 следует заменить директиву stdcall на cdecl:

```
function example1(i1:Integer:i2:Integer):Integer:cdecl:external:
```

Как видим, изменения в исходных текстах минимальны. Рассмотрим теперь самый быстрый способ передачи параметров, использующий соглашение register. В этом случае в вызывающей программе изменения также оказываются минимальными, и касаются они объявления процедуры example1:

```
function example3(i1:Integer:i2:Integer):Integer:register:external:
```

Исходный текст процедуры example1 при использовании соглашения register намного упрощается (листинг 10.3).

**Листинг 10.3.** Модифицированный вариант процедуры example1 для соглашения register

```
.686
.model flat
option casemap:none
.code
```

```
example3 proc
 sub EAX, EDX
 ret
example3 endp
end
```

Поскольку параметры в процедуру при данном соглашении передаются через регистры, то для вычисления разности i1 – i2 нужно выполнить команду

```
sub EAX, EDX
```

Напомню, что соглашение register (см. табл. 10.1) предполагает размещение левого параметра (i1) в регистре EAX, а правого (i2) — в регистре EDX. Поэтому результат можно получить при помощи всего одной команды — sub. Поскольку результат вычитания остается в регистре EAX, то на этом процедура example1 работу заканчивает. Как видно, для такого метода передачи параметров требуется меньшее число команд, поскольку не нужно обращаться к области стека и, кроме того, регистровые операции требуют меньше машинных циклов. По этой причине метод вызова процедур с использованием регистров является очень быстрым.

До сих пор мы рассматривали работу процедур, возвращающих в регистре EAX определенное значение. Намного чаще в вызывающую программу возвращается не сама переменная, а ее адрес. В этом случае говорят, что процедура возвращает ссылку. При помощи ссылки можно обращаться к строкам и массивам, что значительно расширяет область применения ассемблерных процедур.

В следующем примере процедура (назовем ее example2) вычисляет разность двух чисел с плавающей точкой, являющихся ее параметрами, и возвращает адрес результата в вызывающую программу (листинг 10.4). При вызове процедуры используется соглашение stdcall.

**Листинг 10.4.** Вычисление разности двух чисел с плавающей точкой

```
.686
.model flat
option casemap:none
.data
 res DD 0
.code
 example2 proc
  push EBP
  mov  EBP, ESP
  finit
  fld  dword ptr [EBP+8]
  fsub dword ptr [EBP+12]
  fstp dword ptr res
  lea  EAX, res
  pop  EBP
  ret  8
 example2 endp
 end
```

Во многом исходный текст процедуры example2 напоминает программный код процедуры example1 из листинга 10.1, поэтому остановлюсь только на изменениях. Во-первых, здесь используются команды математического сопроцессора для

операций над числами с плавающей точкой, во-вторых, вместо значения в регистре EAX возвращается адрес переменной res, содержащей разность вещественных чисел (команда lea EAX, res).

Код вызывающей программы на Delphi показан в листинге 10.5.

**Листинг 10.5.** Вызывающая программа для процедуры из листинга 10.4

```
program Project1:

{$APPTYPE CONSOLE}

uses
  SysUtils:

{$L f:\example2.obj}

function example2(x1:Single:x2:single):PSingle:stdcall:external:

var
b1, b2: Single:
fres: PSingle:

begin
  b1:= -23.78:
  b2:= -45.09:
  fres:= example2(b1, b2):
  WriteLn(FloatToStr(fres^)):
end
```

Прежде всего, нужно указать компоновщику Delphi, где находится объектный файл с процедурой example2, что выполняется директивой

```
{$L f:\example2.obj}
```

Далее объявляется процедура example2 с соответствующими атрибутами:

```
function example2(x1:Single:x2:Single):PSingle:stdcall:external:
```

Параметрами этой функции выступают числа с плавающей точкой в коротком (Single) формате x1 и x2. Процедура возвращает указатель (PSingle) на результат вычитания x1 из x2. Остальные атрибуты (stdcall и external) мы рассматривали ранее, поэтому останавливаться на них я не буду.

Исходный текст процедуры несложен, но я хочу обратить внимание на два оператора:

```
fres:= example2(b1, b2):
WriteLn(FloatToStr(fres^)):
```

Первый оператор запоминает адрес результата в переменной-указателе fres, а второй выполняет несколько действий:

1. Разыменовывает указатель fres, то есть извлекает значение по указанному адресу (оператор ^).
2. Преобразует полученное число в формате плавающей точки в строку символов (функция FloatToStr).
3. Выводит результат операции на экран дисплея (функция WriteLn).

При указанных значениях переменных b1 и b2 результат, выводимый на экран дисплея, будет равен 21,31.

# 10.3. Интерфейс ассемблерных процедур с Visual C++ .NET 2005

Проанализируем, как будет выглядеть интерфейс ассемблерных процедур с программами, написанными на Visual C++ .NET. Воспользуемся исходными текстами процедур example1 и example2, рассмотренными в предыдущем разделе, для демонстрации принципов построения интерфейса с программами на Visual C++ .NET.

В первом примере при помощи процедуры example1 вычислим разность двух целых чисел и отобразим результат операции на экране дисплея. Предположим, что процедура использует соглашение stdcall. В этом случае исходный текст процедуры example1 останется практически без изменений, но потребуется внести коррективы в имя процедуры: добавить в начале имени символ подчеркивания, а в конце имени — суффикс @*n*, где *n* — число байтов, необходимое для передачи параметров. Подобная форма именования соответствует требованиям компилятора C++ для соглашения stdcall. С учетом этих изменений исходный текст процедуры example1 будет таким, как показано в листинге 10.6.

**Листинг 10.6.** Модифицированная версия процедуры example1 для работы с C++ .NET (соглашение stdcall)

```
.686
.model flat
option casemap:none
.data
  res DD 0
.code
 _example1@8 proc
  push EBP
  mov  EBP, ESP
  finit
  fild  dword ptr [EBP+8]
  fisub dword ptr [EBP+12]
  fistp dword ptr res
  mov   EAX, res
  pop   EBP
  ret  8
 _example1@8 endp
 end
```

Обратите внимание на способ формирования имени ассемблерной процедуры (_example1@8) — он соответствует требованиям соглашения stdcall.

Вызывающая программа на Visual C++ .NET будет такой, как показано в листинге 10.7.

**Листинг 10.7.** Вызывающая программа для процедуры из листинга 10.6

```
#include <stdio.h>
extern "C" int __stdcall example1(int i1, int i2);
int main(void)
{
  int i1 = 455;
  int i2 = -743;
  printf("EXAMPLE1: %d\n", example1(i1, i2));
  return 0;
}
```

В этой программе процедура example1 объявлена следующим образом:

```
extern "C" int __stdcall example1(int i1, int i2);
```

Здесь ключевое слово extern указывает на то, что процедура example1 является внешней, директива __stdcall устанавливает соглашение об именовании и, кроме этого, требует формирования имени вызываемой процедуры специальным образом (как было показано ранее).

Должен заметить, что для компилятора Visual C++ .NET соглашением об именовании по умолчанию является cdecl, поэтому указывать его необязательно. Любое другое соглашение (stdcall, fastcall) необходимо указывать явным образом при объявлении внешней процедуры. Двойное подчеркивание при задании соглашения об именовании обязательно.

Оператор "C" предотвращает декорирование имени процедуры. Этот параметр имеет смысл только для компилятора C++, и мы не будем на нем останавливаться — достаточно помнить, что в объявлении процедуры присутствие оператора "C" необходимо.

Смысл входных параметров процедуры, а также возвращаемого значения понятен и в объяснениях не нуждается. При указанных значениях переменных i1 и i2 результат равен 1198.

Модифицируем предыдущий пример так, чтобы вызываемая процедура использовала соглашение об именовании cdecl. Для этого в исходном тексте процедуры example1 следует изменить имя процедуры, чтобы оно удовлетворяло соглашению cdecl — в начале имени должен присутствовать символ подчеркивания, в то время как оставшаяся часть остается неизменной. Кроме того, в исходном тексте процедуры команду ret нужно указать без параметров. Модифицированный вариант процедуры example1, удовлетворяющий соглашению cdecl, представлен в листинге 10.8.

**Листинг 10.8.** Модифицированная версия процедуры example1, использующая соглашение cdecl

```
.686
.model flat
option casemap:none
.data
  res DD 0
.code
 _example1 proc
```

```
  push EBP
  mov  EBP, ESP
  finit
  fild  dword ptr [EBP+8]
  fisub dword ptr [EBP+12]
  fistp dword ptr res
  mov   EAX, res
  pop   EBP
  ret
 _example1 endp
 end
```

Программный код вызывающей программы на Visual C++ .NET показан в листинге 10.9.

**Листинг 10.9.** Вызывающая программа для процедуры из листинга 10.8

```
#include <stdio.h>
extern "C" int example1(int i1, int i2);
int main(void)
{
 int i1 = 45;
 int i2 = -73;
 printf("EXAMPLE1: %d\n", example1(i1, i2));
 return 0;
}
```

Обратите внимание на объявление внешней процедуры example1. Здесь отсутствует явное указание соглашения о передаче параметров, поскольку по умолчанию используется соглашение cdecl.

Проведем анализ самого быстрого способа передачи параметров — fastcall. В соответствии с табл. 10.1 первые два параметра в вызываемую процедуру передаются слева направо с помощью регистров ECX и EDX, а остальные — справа налево через стек. Кроме того, имя вызываемой процедуры при таком соглашении должно формироваться следующим образом: в начале имени процедуры ставится знак амперсанда (@), а в конце — сочетание символов @*n*, имеющее тот же смысл, что и для соглашения stdcall. Исходный текст процедуры example1, удовлетворяющий соглашению fastcall, показан в листинге 10.10.

**Листинг 10.10.** Модифицированная версия процедуры example1, использующая соглашение fastcall

```
.686
.model flat
option casemap:none
.code
 @example1@8 proc
  mov   EAX, ECX
  sub   EAX, EDX
  ret
 @example1@8 endp
 end
```

Вызывающая программа на Visual C++ .NET показана в листинге 10.11.

**Листинг 10.11.** Вызывающая программа для процедуры из листинга 10.10

```
#include <stdio.h>
extern "C" int __fastcall example1(int i1. int i2);
int main(void)
{
  int i1 = 145;
  int i2 = -203;
  printf("EXAMPLE1: %d\n", example1(i1, i2));
  return 0;
}
```

До сих пор все рассматриваемые версии процедуры example1 возвращали в вызывающую программу непосредственное значение. Сейчас мы проанализируем пример процедуры, возвращающей не значение переменной, а указатель на значение (адрес). В листинге 10.12 приводится исходный текст процедуры example2, вычисляющей разность двух чисел с плавающей точкой, которые являются входными параметрами для этой процедуры. Процедура возвращает в регистре EAX адрес результата. Будем полагать, что для процедуры example2 принято соглашение об именовании stdcall.

**Листинг 10.12.** Вычисление разности двух чисел с плавающей точкой

```
.686
.model flat
option casemap:none
.data
  res DD 0
.code
 _example2@8 proc
  push EBP
  mov  EBP, ESP
  finit
  fld  dword ptr [EBP+8]
  fsub dword ptr [EBP+12]
  fstp dword ptr res
  lea  EAX, res
  pop  EBP
  ret  8
 _example2@8 endp
 end
```

Программный код вызывающей программы на Visual C++ .NET показан в листинге 10.13.

**Листинг 10.13.** Вызывающая программа для процедуры из листинга 10.12

```
#include <stdio.h>
extern "C" float* __stdcall example2(float f1, float f2);
int main(void)
{
 float f1 = 1.45;
 float f2 = -2.03;
 printf("\nEXAMPLE2: %5.2f\n", *example2(f1, f2));
 return 0;
}
```

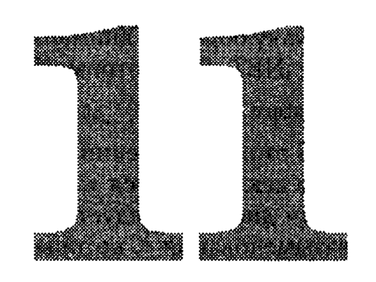
Хочу обратить внимание читателей на то, как обрабатываются ссылки в Visual C++ .NET. Рассмотрим объявление процедуры example2:

```
extern "C" float* __stdcall example2(float f1, float f2);
```

В этом объявлении указатель на число с плавающей точкой в коротком формате декларируется как float*. Для получения значения числа и вывода его на экран дисплея следует разыменовать указатель, для чего используется оператор разыменования *, который должен помещаться перед указателем. Следующий оператор C++ демонстрирует это:

```
printf("\nEXAMPLE2: %5.2f\n", *example2(f1, f2));
```

# Процессоры Intel Pentium в современных разработках



Эта глава посвящена анализу вычислительных возможностей последних поколений процессоров Intel Pentium для платформы IA-32, которые получили широкое распространение как в промышленных системах, так и в домашних персональных компьютерах. Мы рассмотрим особенности использования процессоров Intel Pentium 4 и начнем с обзора микроархитектуры NetBurst, на которой и базируется это поколение процессоров.

## 11.1. Микроархитектура Intel NetBurst

Основные особенности NetBurst:

- гиперконвейерная (hyper-pipelined) технология;
- применение кэша трассировки выполнения команд (execution trace cache);
- повышенная частота (400 МГц) системной шины;
- улучшенная схемотехническая и аппаратная реализация модулей целочисленных арифметических операций (rapid execution engine);
- использование специального алгоритма улучшенного динамического выполнения команд (advanced dynamic execution);
- использование потокового SSE2-расширения;
- улучшенный кэш передачи данных (advanced transfer cache);
- аппаратная реализация блока предварительной выборки (выборки с упреждением) команд (hardware prefetcher).

Рассмотрим особенности NetBurst более подробно:

- В гиперконвейере команд используется 20-ступенчатая выборка, что в два раза превышает глубину очереди для процессоров линейки P6. Это позволяет

обеспечивать высокую производительность и, кроме того, способствует дальнейшему увеличению тактовой частоты.

- Кэш трассировки выполнения команд обеспечивает принципиально новую реализацию кэша команд 1-го уровня. Он кэширует команды процессора (микрооперации), устраняя задержки, создаваемые декодером команд (instruction decoder). Кроме того, кэш трассировки выполнения команд сохраняет результаты ветвлений в одной и той же линейке кэша, что позволяет повышать скорость передачи команд из кэша и приводит к лучшему использованию пространства памяти, который он занимает (12 К микроопераций). В общем итоге применение кэша трассировки позволяет увеличить количество инструкций, обрабатываемых исполнительными модулями процессора, и уменьшить время восстановления очереди команд после неправильно спрогнозированных переходов.
- Частота обмена данными 400 МГц вместе с интерфейсом системной шины и схемами умножения позволяет обеспечивать прием/передачу данных в процессор на скорости, превышающей 3,2 Гбайт/с.
- Улучшенная схемотехническая и аппаратная реализация модулей целочисленных арифметических операций обеспечивает работу арифметико-логических модулей (Arithmetic Logic Units, ALUs) центрального процессора на частоте, превышающей в два раза частоту базового кристалла процессора. Это позволяет выполнять некоторые инструкции за половину такта частоты процессора, что в общем итоге приводит к повышению производительности и уменьшению задержек при выполнении команд.
- Специальный алгоритм улучшенного динамического выполнения команд обеспечивает спекулятивное (условное) выполнение инструкций процессора в окне (буфере памяти), содержащем до 126 инструкций. Такой большой размер окна позволяет избегать задержек при выполнении инструкций, ожидающих результатов выполнения других команд (например, команд получения данных из памяти), за счет изменения порядка выполнения находящихся в этом окне инструкций. В общем итоге это повышает загрузку центрального процессора. Технология улучшенного динамического выполнения команд обеспечивает более совершенное предсказание ветвлений, что уменьшает ошибки предсказаний приблизительно на 33 % по сравнению с архитектурой P6. Это достигается не в последнюю очередь и за счет использования буфера предсказаний (Branch Target Buffer, BTB) размером 4 Кбайт, который позволяет сохранять более детальную историю уже выполненных ветвлений.
- Микроархитектура NetBurst расширяет возможности технологий MMX и SSE за счет добавления 144 новых команд потокового SSE2-расширения, предназначенных для выполнения операций над 128-разрядными целочисленными данными и данными с плавающей точкой двойной точности. Новые команды обеспечивают более высокую производительность при разработке программ для процессора Intel Pentium 4. Использование SSE2-расширения значительно повышает производительность приложений. Мы будем рассматривать практические аспекты применения технологии SSE2 в главе 14.

- Улучшенный кэш передачи данных — это вспомогательный кэш размером 256 Кбайт, поддерживающий высокоскоростной канал передачи данных между кэшем 2-го уровня и процессором. Он включает в себя 256-разрядный (32-байтовый) интерфейс, выполняющий передачу данных за один такт частоты процессора. Предположим, что процессор Intel Pentium 4 работает на частоте 2,4 ГГц. В этом случае скорость передачи данных достигает 76,8 Гбайт/с (32 байта × 2,4 ГГц = 76,8 Гбайт/с). Использование такого кэша обеспечивает высокий процент загрузки центрального процессора.
- При аппаратной реализации блока предварительной выборки (выборки с упреждением) команд, впервые реализованной в процессорах Intel Pentium 4, этот блок представляет собой отдельный аппаратный модуль процессора, прозрачный по отношению к выполняемым программам. Он обеспечивает упреждающую выборку команд на основе анализа вычислительного алгоритма программы. Такое решение позволяет снижать задержки при обращении к оперативной памяти в процессе выполнения программы, что повышает производительность работы приложения.

## 11.2. Особенности работы приложений с процессором Intel Pentium 4

Производительность выполнения приложений для процессоров, работающих на высоких частотах, может варьироваться в значительных пределах. Это связано, главным образом, с тем, что для каждого приложения пишется свой уникальный программный код. Диапазон разрабатываемых приложений весьма широк: от простых офисных программ до сложных мультимедийных проектов.

Одним из показателей производительности программ является количество команд, или инструкций, процессора, выполняемых в единицу времени. Этот показатель в значительной степени зависит от числа условных переходов и ветвлений в программе, а также от предсказуемости этих переходов и ветвлений. Чем больше программный код содержит непредсказуемых или плохо предсказуемых ветвлений, тем больше времени процессор будет тратить на бесполезную работу. Например, в приложениях, интенсивно обрабатывающих целые числа, а также в офисных приложениях, таких, как текстовые процессоры, обычно выполняется много переходов и ветвлений, что снижает производительность их работы.

В общем случае, производительность выполнения приложений с большим числом ветвлений не зависит линейно от частоты используемого процессора и меняется относительно медленно, несмотря на архитектурные улучшения современных процессоров, такие, например, как многоступенчатые конвейеры команд.

В то же время в мультимедийных приложениях, а также в приложениях, в которых выполняются в основном операции над числами с плавающей точкой, ветвления и переходы, как правило, предсказуемы с высокой степенью вероятности. По этой причине такие приложения показывают более высокую производительность по сравнению с целочисленными и офисными приложениями.

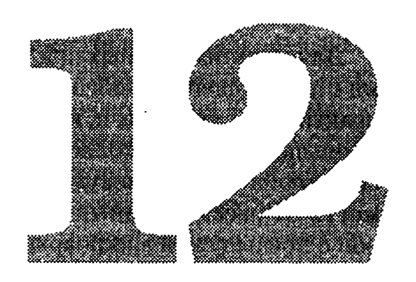
Для мультимедийных приложений, как и для приложений, оперирующих числами с плавающей точкой, зависимость производительности от частоты процессора практически линейна, и, кроме того, здесь ощущается преимущество процессоров с большей глубиной конвейера команд. Процессор Intel Pentium 4 позволяет повысить производительность работы приложений, хотя и в разной степени, в зависимости от рассмотренных выше условий. В общем случае, переход от процессоров с предыдущей микроархитектурой, например Intel Pentium III, к процессору Intel Pentium 4 не повышает производительность работы приложений во столько же раз, во сколько возрастает частота процессора.

Для того чтобы воспользоваться преимуществами архитектуры новых процессоров, таких, как Intel Pentium 4, можно перекомпилировать приложение при помощи более новой версии компилятора либо задействовать оптимизированные библиотеки функций для работы с мультимедийными MMX-, SSE- или SSE2-расширениями. В общем, оптимизация приложений для процессоров Intel Pentium может идти в трех направлениях:

- конвейеризация и буферизация памяти;
- разработка эффективных вычислительных алгоритмов;
- эффективное использование системной шины.

В следующих главах мы акцентируем внимание на разработке эффективных алгоритмов обработки данных с использованием технологии SIMD, а точнее — MMX-, SSE- и SSE2-расширений процессоров Intel Pentium.

# MMX-расширение процессоров Intel Pentium



Материал этой главы открывает обзор наиболее мощных технологий обработки данных, разработанных фирмой Intel и поддерживаемых последними поколениями процессоров Pentium. Эта группа технологий известна под названием SIMD (Single Instruction, Multiple Data — одна команда, много данных). Технологии SIMD представляют собой расширения базовой архитектуры IA-32 процессоров Intel и включают в себя дополнительные регистры, типы данных и команды. Основная цель включения этих расширений в архитектуру Intel — добиться более высокой производительности работы мультимедийных приложений, а также систем обработки и передачи данных. В практическом плане SIMD реализована как две взаимосвязанные технологии обработки данных:

- с помощью технологии MMX (MultiMedia eXtensions — мультимедийные расширения) выполняется высокоэффективная обработка данных целочисленного типа, имеющих разрядность 64 бита;
- технология SSE (Streaming SIMD Extensions — потоковые SIMD-расширения) предназначена для эффективной обработки данных вещественного типа с разрядностью 128 бит.

Эти технологии позволяют разработать высокопроизводительные приложения при решении следующих задач:

- кодирование, декодирование и обработка сигналов;
- распознавание речи;
- обработка и захват видеосигналов;
- манипулирование объектами 3D-графики;
- обработка 3D-звука;
- промышленное проектирование (CAD/CAM).

Главное преимущество архитектуры SIMD заключается в том, что многие вычисления можно выполнять одновременно или, как говорят, параллельно над несколькими операндами, что позволяет увеличить быстродействие программного кода.

Применение ассемблера в SIMD-расширениях позволяет писать компактный и быстрый код для критических фрагментов приложения с использованием специальных инструкций процессора для SIMD-расширений.

Материал этой главы посвящен анализу операций MMX-расширения, позволяющего обрабатывать целочисленные данные параллельно. Несмотря на широкое распространение, которое получила технология MMX, информация, касающаяся практических аспектов ее применения, во многих случаях не систематизирована или отрывочна. Представленный в этой главе материал призван восполнить этот пробел. Здесь вы найдете довольно подробную информацию о данной технологии и о ее применении, но вначале проанализируем некоторые принципиальные вопросы реализации технологии MMX.

В основе технологии MMX лежит расширение набора команд процессора с учетом потребностей современных мультимедийных программ. Команды технологии MMX работают с новыми типами данных: 64-разрядными целочисленными данными, а также с данными, упакованными в группы общей длиной 64 бита. Такие данные могут находиться в памяти или в восьми MMX-регистрах, которые обозначаются как `MM0` – `MM7`.

Технология MMX расширяет функциональные возможности процессоров архитектуры Intel при полной совместимости с программами, написанными для предыдущих поколений процессоров. Все подобные программы безо всяких изменений могут выполняться на процессорах, поддерживающих технологию MMX. Команды MMX-расширения выполняются так же, как и команды с плавающей точкой. Более того, механизм сохранения и восстановления состояния вычислительной среды, принятый для операций с плавающей точкой, применим и при выполнении MMX-вычислений. Технология MMX не требует какого-то специального режима работы процессора или дополнительных аппаратных ресурсов — команды MMX-расширения выполняются в том же режиме процессора, что и команды с плавающей точкой.

Команды MMX-расширения обеспечивают параллельную обработку нескольких байтов, слов или двойных слов, а также поддерживают работу со следующими типами данных:

- упакованные байты (packed byte) — один 64-разрядный регистр содержит 8 байт (рис. 12.1);

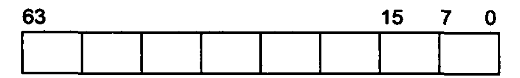



**Рис. 12.1.** 8-байтовый формат представления данных

- упакованные слова (packed word) — один 64-разрядный регистр содержит четыре 16-разрядных слова (рис. 12.2);

63 15 0

**Рис. 12.2.** Формат представления данных в виде четырех слов

- упакованные двойные слова (packed doubleword) — один 64-разрядный регистр содержит два 32-разрядных слова (рис. 12.3);

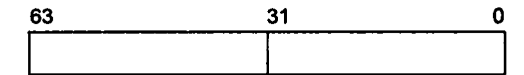



**Рис. 12.3.** Формат представления данных в виде двух двойных слов

- 64-разрядные слова (quadword) (рис. 12.4).

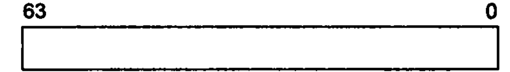



**Рис. 12.4.** Формат представления данных в виде учетверенного слова

При работе с MMX-командами используются регистры стека математического сопроцессора R0 – R7. При этом вместо 80 бит задействуются 64, а стековая организация, требуемая для операций сопроцессора, не используется. Регистровый стек в операциях MMX-расширения рассматривается как группа из восьми независимых 64-разрядных регистров (рис. 12.5).

Стек регистров сопроцессора — MMX-регистры

79 63 0 — 63 0

R0 → MM0
R1 → MM1
R2 → MM2
R3 → MM3
R4 → MM4
R5 → MM5
R6 → MM6
R7 → MM7

**Рис. 12.5.** Соответствие регистров сопроцессора и MMX-расширения

Хочу упомянуть одно важное правило, которое следует соблюдать при совместном использовании математического сопроцессора и MMX-расширения: последней выполняемой командой MMX-расширения должна быть команда emms.

Дело в том, что все MMX-команды выполняются в том же режиме процессора, что и команды с плавающей точкой, что вызывает изменения содержимого регистра состояния (swr) сопроцессора. Команда emms обеспечивает корректный переход процессора от выполнения фрагмента программного кода с MMX-командами к обработке обычных команд с плавающей точкой. При этом emms устанавливает значение 1 во всех разрядах регистра состояния. Если фрагмент программы, в котором есть MMX-команды, не заканчивается командой emms, то все последующие операции с плавающей точкой будут давать некорректные результаты, о чем сигнализирует исключение Stack overflow.

Большинство команд MMX-расширения имеют следующий синтаксис:

```
имя_команды приемник, источник
```

Здесь *приемник* — это выходной операнд, или операнд-приемник, а *источник* — входной операнд, или операнд-источник. Обычно входная информация извлекается из обоих операндов, а результат записывается в выходной операнд.

Обработка данных MMX-расширения может выполняться одним из двух способов: с использованием либо циклической арифметики (wraparound arithmetic), либо арифметики с насыщением (saturation arithmetic). Большинство команд технологии MMX обрабатывают данные по правилам циклической арифметики, а некоторые команды задействуют арифметику с насыщением.

Если команда задействует циклическую арифметику (другое название — арифметика с циклическим переносом) и результат операции выходит за двоичную разрядную сетку используемого типа данных, то «лишние» старшие биты результата отбрасываются. Иначе говоря, если результат превышает максимально возможное значение на *n* единиц, то результатом считается минимальное значение плюс *n* и минус 1. Например, сложение байтов 01h и FFh дает 00h.

Если команда использует арифметику с насыщением и результат операции превышает максимальное представимое значение, то в выходной операнд записывается это максимальное значение (происходит «насыщение»). Аналогично, если результат операции оказывается меньше нижней границы допустимого диапазона, то в выходной операнд записывается минимальное возможное значение. Например, если результат меньше 8000h, то 16-разрядное слово со знаком считается равным 8000h. Если полученное значение больше 7FFFh, то слово со знаком считается равным 7FFFh.

В арифметике с насыщением MMX-команды сложения, вычитания и упаковки данных могут обрабатывать числа со знаком или без знака. Данные со знаком и без знака имеют различный допустимый диапазон. Следовательно, если используется арифметика с насыщением, то при выходе результата операции за пределы допустимого диапазона в выходной операнд записываются различные значения, в зависимости от типа данных. Например, если результат превышает 7FFFh, слово со знаком считается равным 7FFFh, а слово без знака — нет.

Вернемся к анализу синтаксиса MMX-команд. Большинство команд имеют суффикс, который определяет тип данных и используемую арифметику:

- us (unsigned saturation) — арифметика с насыщением, данные без знака или, по-другому, беззнаковое насыщение. Если команда использует арифметику с насыщением и результат операции превышает максимальное представимое значение, то в выходной операнд записывается это максимальное значение (происходит «насыщение»). Аналогично, если результат операции оказался меньше нижней границы допустимого диапазона, то в выходной операнд записывается минимально возможное значение;
- s или ss (signed saturation) — арифметика с насыщением, данные со знаком или, по-другому, знаковое насыщение;

- если в суффиксе нет ни символа s, ни символов ss, то применяется циклическая арифметика (wraparound). Если в этом случае результат операции выходит за двоичную разрядную сетку используемого типа данных, то «лишние» старшие биты результата отбрасываются;
- b, w, d, q — эти буквы указывают тип данных. Если в суффиксе есть две из этих букв, первая соответствует входному операнду, вторая — выходному.

Вот некоторые примеры. Следующая команда выполняет сложение слов без знака:

```
paddusw MM0.meml
```

Здесь суффикс us означает, что в команде используется арифметика с насыщением без знака, а операнды имеют разрядность слов. Первое слагаемое находится в MMX-регистре MM0, а второе — в памяти по адресу meml. Результат сохраняется в регистре MM0.

Еще один пример:

```
pand MM0. MM1
```

В этой команде регистр MM0 является входным операндом, а регистр MM1 — выходным. Команда вычисляет поразрядное логическое И значений, содержащихся в регистрах MM0 и MM1, и сохраняет результат в регистре MM0.

Перейдем к анализу MMX-команд. Их условно можно разделить на группы:

- команды сложения и вычитания;
- команды сдвига;
- логические команды;
- команды умножения;
- команды сравнения;
- команды упаковки и распаковки;
- команды передачи данных.

Познакомимся с каждой группой команд более подробно и начнем с команд передачи данных.

## 12.1. Команды передачи данных

В группу команд передачи данных входят команды movd и movq. Команда movd позволяет копировать 32-разрядное число:

- из младших разрядов одного MMX-регистра в младшие разряды другого (старшие разряды заполняются нулями);
- из переменной в памяти либо из целочисленного регистра в младшие 32 разряда MMX-регистра (старшие разряды заполняются нулями);
- из младших разрядов MMX-регистра в ячейку памяти либо в целочисленный регистр.

Команда movq выполняет копирование 64 бит:

- из одного MMX-регистра в другой;
- из памяти в MMX-регистр;
- из MMX-регистра в память.

Среди всех MMX-команд только movd и movq могут иметь выходной операнд в памяти, а movd — единственная команда, операнд которой может находиться в 32-разрядном регистре процессора.

## 12.2. Команды сложения

MMX-команды сложения и вычитания работают с упакованными байтами и словами со знаком и без знака, а также с упакованными двойными словами со знаком. Они могут использовать как циклическую арифметику, так и арифметику с насыщением. К командам этой группы относятся:

- paddb, paddw, paddd — формируют результат по принципу циклической арифметики. Команды padd выполняют сложение элементов данных (байтов, слов или двойных слов) входного и выходного операндов. Если сумма выходит за границу допустимого диапазона, то, по правилам циклической арифметики, избыток отсчитывается от другой границы диапазона. «Переноса» единицы из одного элемента данных в другой не происходит. Входной операнд может находиться в MMX-регистре или в памяти; выходной операнд должен находиться в MMX-регистре;
- paddsb, paddsw — формируют результат по принципу арифметики со знаковым насыщением. Команды выполняют сложение элементов данных (байтов или слов) входного и выходного операндов. Если сумма выходит за граничное значение допустимого диапазона, то результатом считается это граничное значение. Входной операнд может находиться в MMX-регистре или в памяти; выходной операнд должен находиться в MMX-регистре;
- paddusb, paddusw — формируют результат по принципу арифметики с беззнаковым насыщением. Команды выполняют сложение элементов данных (байтов или слов) входного и выходного операндов. Если сумма выходит за пределы граничного значения из допустимого диапазона, то результатом считается это граничное значение. Входной операнд может находиться в MMX-регистре или в памяти; выходной операнд должен находиться в MMX-регистре.

Работу команды paddusw при сложении слов по принципу беззнакового насыщения иллюстрирует рис. 12.6.

Из рисунка видно, что выходным операндом (операндом-приемником) команды paddusw является регистр MM0, причем второе слово (считая с нуля по возрастанию) содержит значение 65 535. Такой результат является следствием того, что сумма вторых слов превышает предельно допустимое значение для данного типа операндов (16-разрядное слово без знака), поэтому в качестве суммы берется граничное значение.

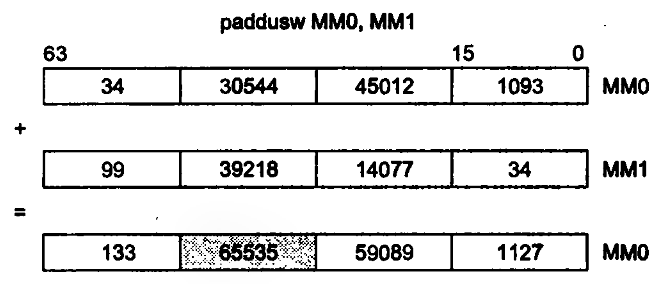



**Рис. 12.6.** Сложение беззнаковых слов с насыщением

Продемонстрирую работу команд сложения на примерах, но вначале уточню некоторые аспекты разработки программ, в которых используются MMX-команды. Для разработки таких программ подходят только последние версии компиляторов MASM, например компилятор версии 7.10*.xxxx*, входящий в состав Windows XP DDK или Windows Server 2003 DDK. Кроме того, в исходных текстах программ обязательно должна присутствовать директива `.MMX`, чтобы компилятор мог транслировать MMX-команды в машинные коды.

Все примеры этой главы реализованы в виде отдельных 32-разрядных процедур. Большинство процедур содержит область данных с переменными, заданными определенными значениями. Это позволяет легко проверить результаты вычислений и отобразить их на экране, для чего используются простейшие консольные программы, написанные на языке Visual C++ .NET, хотя можно применить и другой 32-разрядный компилятор без переделки исходных текстов.

Перед тем как создавать программы, использующие MMX-расширения, следует убедиться в том, что данный тип процессора поддерживает эту технологию. Для этого можно выполнить ассемблерную команду `cpuid`, предварительно поместив в регистр `EAX` значение 1. После выполнения команды проверка 23-го бита в регистре `EDX` показывает, поддерживается ли технология MMX процессором. Если этот бит установлен в единицу, то поддерживается.

В листинге 12.1 приводится исходный текст процедуры на ассемблере, позволяющей определить, поддерживается ли МММ-расширение данным процессором.

**Листинг 12.1.** Проверка поддержки процессором MMX-расширения

```
.686
.model flat
option casemap: none
.data
  supMMX DB 1
.code
 _test_mmx proc
   mov   EAX, 1
   cpuid
   test  EDX, 800000h
   jnz   exit
   mov   supMMX, 0
 exit:
   xor   EAX, EAX
```

```
   mov   AL, supMMX
   ret
 _test_mmx endp
 end
```

Процедура _test_mmx возвращает 1 в регистре AL, если технология MMX поддерживается процессором, и 0 — в противном случае.

Наш первый пример демонстрирует выполнение команды paddb и реализован в виде процедуры _paddb_ex (листинг 12.2).

**Листинг 12.2.** Сложение байтов с использованием команды paddb

```
.686
.model flat
.MMX
option casemap:none
.data
 src  DB "PHILADELPHIA FLYERS"
 len  EQU $-src
 tmp  DB len DUP (20h)
 dst  DB len DUP(' '),0
.code
 _paddb_ex proc
   mov   EAX, len
   mov   EBX, 8
   xor   EDX, EDX
   div   EBX
   mov   ECX, EAX
   lea   ESI, src
   lea   EDI, dst
   lea   EBX, tmp
next:
   movq  MM0, qword ptr [ESI]
   paddb MM0, qword ptr [EBX]
   movq  qword ptr [EDI], MM0
   add   ESI, 8
   add   EDI, 8
   add   EBX, 8
   dec   ECX
   jnz   next
   cmp   EDX, 0
   jz    exit
   mov   ECX, EDX
next1:
   mov   AL, byte ptr [ESI]
   add   AL, 20h
   mov   byte ptr [EDI], AL
   inc   ESI
   inc   EDI
   dec   ECX
   jnz   next1
 exit:
   lea   EAX, dst
   ret
 _paddb_ex endp
 end
```

Процедура _paddb_ex заменяет прописные символы (байты) строки src строчными. Для этого к каждому символу строки src прибавляется значение 20h, после чего модифицированный символ сохраняется в области памяти dst. Процедура возвращает в вызывающую программу адрес модифицированной строки dst в регистре EAX. Напомню, что команда paddb выполняет одновременное сложение 8 байт источника и 8 байт приемника, поэтому для использования этой команды нужно выделить из строки src группы символов по 8 байт. Если остается меньше 8 символов, то операции с ними можно выполнить с помощью обычных команд. Например, в нашем случае строка src содержит 19 символов, или 8 × 2 + 3. Следовательно, 16 символов можно обработать двукратным вызовом команды paddb, а оставшиеся 3 — с помощью обычных команд. Если бы обрабатываемая строка содержала, например, 35 символов, то ее обработку можно было бы выполнить за счет 4 вызовов команды paddb и, кроме этого, дополнительно обработать еще 3 символа.

Посмотрим на равенство 8 × 2 + 3 = 19. Число 2 в этой формуле определяет возможное количество итераций цикла (счетчик итераций) для команды paddb, то есть для заданной строки src выполнится 2 итерации. Следующая группа команд позволяет определить значение счетчика итераций и поместить его в регистр ECX:

```
mov   EAX, len
mov   EBX, 8
xor   EDX, EDX
div   EBX
mov   ECX, EAX
```

После выполнения этой группы команд регистр ECX будет содержать счетчик итераций (в данном случае он равен 1), а регистр EDX — количество символов, которые необходимо обработать вне основного цикла с помощью обычных команд. Напомню, что при делении содержимого двух 32-разрядных регистров (EDX:EAX в данном случае) на содержимое 32-разрядного регистра EBX частное сохраняется в регистре EAX, а остаток — в регистре EDX.

Следующая группа команд загружает адреса обрабатываемых строк:

```
lea   ESI, src
lea   EDI, dst
lea   EBX, tmp
```

Здесь src — строка, которая обрабатывается в данный момент, переменная dst указывает на строку, в которой будет содержаться результат, а tmp представляет строку, содержащую группу байтов, каждый из которых равен 20h. Строки src и tmp имеют одинаковый размер, а строка dst — на 1 больше, поскольку содержит символ 0. Наша процедура возвращает результат в виде адреса строки dst, поэтому вызывающая программа, обнаружив 0, определяет конец строки.

Собственно сложение байтов строк src и tmp выполняется при помощи команд.

```
movq  MM0, qword ptr [ESI]
paddb MM0, qword ptr [EBX]
```

А сохранение результата — с помощью команды

```
movq qword ptr [EDI], MM0
```

Обратите внимание на то, что в этих командах используется спецификатор qword, указывающий на учетверенное слово (8 байт). Команды emms здесь не требуется, поскольку никакие команды сопроцессора не используются, и синхронизация не нужна.

Переход к следующей 8-байтовой группе выполняется командами.

```
add   ESI, 8
add   EDI, 8
add   EBX, 8
```

Эти команды продвигают указатели адресов всех строк на 8. Далее проверяется значение счетчика в регистре ECX, и, если он равен 0, происходит выход из цикла. Если значение ECX положительно, выполняется переход к следующей итерации (метка next).

Если цикл завершен (ECX = 0), проверяется содержимое регистра EDX. Вспомним, что EDX содержит число элементов строки src, требующих отдельной обработки. Если это число равно 0, то есть число элементов строки src кратно 8, то происходит выход из процедуры, иначе в счетчик ECX помещается содержимое EDX (в нашем случае — 3) и обработка строки продолжается, но уже обычными командами. Эти действия выполняются в следующем фрагменте кода:

```
cmp   EDX, 0
   jz    exit
   mov   ECX, EDX
next1:
   mov   AL, byte ptr [ESI]
   add   AL, 20h
   mov   byte ptr [EDI], AL
   inc   ESI
   inc   EDI
   dec   ECX
   jnz   next1
 exit:
```

Наконец, предпоследняя команда помещает адрес строки результата в регистр EAX:

```
lea EAX, dst
```

После этого происходит выход из процедуры.

Мы не напрасно анализировали эту процедуру столь подробно. Подобная техника параллельной обработки данных будет использоваться и в последующих примерах. Эта процедура (как и все остальные) является демонстрационной, поэтому для большей наглядности исходные данные содержатся в самом коде, а не передаются в виде параметров. Программный код очень легко модифицировать и использовать как в программах на ассемблере, так и в приложениях на языках высокого уровня.

Хочу добавить, что технология MMX дает выигрыш в производительности по сравнению с использованием обычных команд. Особенно это заметно при обработке больших объемов данных. Если бы наша строка содержала, к примеру, несколько тысяч символов, разница в производительности по сравнению с аналогичной программой, использующей обычные команды, была бы весьма ощутимой.

Для тестирования процедуры _paddb_ex можно написать несложную программу на Visual C++ .NET, которая отображает результат обработки строки на экране (листинг 12.3).

**Листинг 12.3.** Демонстрационная программа для процедуры из листинга 12.2

```
#include <stdio.h>
extern "C" char* paddb_ex(void);
int main(void)
{
 printf("PADDB resut:\n");
 printf("%s\n", paddb_ex());
 return 0;
}
```

В этой программе процедура paddb_ex объявлена внешней, возвращающей результат в виде адреса строки (спецификатор char*). Программа отображает иа экране строку

```
PADDB result:
philadelphia@flyers
```

Рассмотрим более сложный пример, в котором показана работа команды paddw. Процедура _paddw_ex выполняет попарное сложение элементов целочисленных массивов беззнаковых слов и возвращает адрес массива, содержащий суммы элементов, в регистре EAX (листинг 12.4).

**Листинг 12.4.** Сложение целых чисел при помощи команды paddw

```
.686
.model flat
.MMX
option casemap:none
.data
 a1  DW 12094, 31890, 4107, 41499, 32054, 35901, 45033, 50501, 33801
 a2  DW 43701, 39109, 43771, 29507, 47199, 2894, 34722, 23017, 7456
 len EQU $-a2
 dst DW len DUP(0)
.code
 _paddw_ex proc
   mov   EAX, len
   shr   EAX, 1
   mov   EBX, 4
   xor   EDX, EDX
   div   EBX
   mov   ECX, EAX
   lea   ESI, a1
   lea   EDI, a2
   lea   EBX, dst
next:
   movq  MM0, qword ptr [ESI]
   paddw MM0, qword ptr [EDI]
   movq  qword ptr [EBX], MM0
   add   ESI, 8
   add   EDI, 8
```

```
  add  EBX, 8
  dec  ECX
  jnz  next
  cmp  EDX, 0
  jz   exit
  mov  ECX, EDX
next1:
  mov  AX, word ptr [ESI]
  add  AX, word ptr [EDI]
  mov  word ptr [EBX], AX
  add  ESI, 2
  add  EDI, 2
  add  EBX, 2
  dec  ECX
  jnz  next1
 exit:
  lea  EAX, dst
  ret
 _paddw_ex endp
 end
```

Программный код процедуры _paddw_ex начинается с подсчета числа 8-словных групп в массиве a2 целых чисел и определения оставшихся элементов. Это делается с помощью команд

```
mov  EAX, len
shr  EAX, 1
mov  EBX, 4
xor  EDX, EDX
div  EBX
mov  ECX, EAX
```

Здесь команда shr EAX, 1 приводит размерность массива в байтах к размерности слова. После выполнения команды div EBX в регистре EAX содержится количество 64-разрядных элементов массива, а в регистре EDX — количество элементов, требующих обычной обработки. При указанном количестве элементов массивов a1 и a2 (оно равно 9) в регистре EAX содержатся два 64-разрядных элемента, а в регистре EDX — 1 слово. Далее в регистры ESI, EDI и EBX загружаются адреса массивов a1, a2 и dst:

```
lea  ESI, a1
lea  EDI, a2
lea  EBX, dst
```

Сложение 64-разрядных элементов массивов a1 и a2 выполняется командами

```
movq  MM0, qword ptr [ESI]
paddw MM0, qword ptr [EDI]
```

Результаты сложения копируются в массив dst командой

```
movq qword ptr [EBX], MM0
```

После этого указатели адресов продвигаются к следующим элементам массивов:

```
add  ESI, 8
add  EDI, 8
add  EBX, 8
```

Затем цикл повторяется. По окончании цикла анализируется содержимое регистра EDX, и если оно равно 0, то происходит выход из процедуры. Если EDX содержит значение, отличное от нуля, выполняется суммирование оставшихся элементов массивов с помощью обычных команд в цикле:

```
nextl:
   mov   AX, word ptr [ESI]
   . . .
   dec   ECX
   jnz   nextl
```

По окончании операции сложения элементов массивов процедура сохраняет адрес массива dst с результатами вычислений при помощи команды

```
lea EAX, dst
```

Визуально результаты вычислений можно получить с помощью простой программы на Visual C++ .NET, вызывающей процедуру _paddw_ex (листинг 12.5).

**Листинг 12.5.** Демонстрационная программа для процедуры из листинга 12.4

```
#include <stdio.h>
extern "C" unsigned short int* paddw_ex(void);
int main(void)
{
 unsigned short int* paddw = paddw_ex();
 printf("PADDW result:\n");
 for (int i1 = 0;i1 < 9; i1++)
 {
  printf("%d ", *paddw++);
 }
return 0;
}
```

Хочу обратить внимание читателей на следующие особенности вызывающей программы: во-первых, процедура paddw_ex объявлена как внешняя, во-вторых, она возвращает указатель типа unsigned short int. Этот тип указателя используется для адресации беззнакового целого 16-разрядного числа (диапазон изменения 0–65 535), что соответствует типу возвращаемого процедурой paddw_ex значения.

Проанализируем результаты выполнения программы:

```
PADDW result:
55795 5463 47878 5470 13717 38795 14219 7982 41257
```

Для удобства анализа изобразим слагаемые и результат сложения следующим образом:

```
a1 →      12094 31890 4107 41499 32054 35901 45033 50501 33801
+
a2 →      43701 39109 43771 29507 47199 2894 34722 23017 7456
-----------------------------------------------------------
a1 + a2 → 55795 5463 47878 5470 13717 38795 14219 7982 41257
```

Обратите внимание на результаты сложения, выделенные жирным шрифтом, — они получены с помощью команды paddw. Команда paddw формирует результат по принципу циклического переноса, то есть при превышении верхнего значения (65 535) для беззнакового целого числа старшие биты результата отсекаются. Вот почему в результате сложения чисел 31 890 и 39 109 вместо 70 999 мы получили

значение 5463 — это как раз и есть разность между 70 999 и 65 536. То же самое касается и остальных результатов, выделенных жирным шрифтом.

Для демонстрации сложения беззнаковых операндов с помощью команды paddusw рассмотрим еще один пример. Особенностью команды paddusw является то, что она формирует результат по принципу беззнакового насыщения. В листинге 12.6 с помощью процедуры _paddusw_ex выполняется сложение элементов двух целочисленных массивов, a1 и a2. Элементы массивов представляют собой числа без знака размером в слово.

**Листинг 12.6.** Сложение целых чисел с помощью команды paddusw

```
.686
.model flat
.MMX
option casemap:none
.data
 a1    DW 12094, 31890, 4107, 41499, 32054, 35901, 45033
 a2    DW 43701, 39109, 43771, 29507, 7199, 2894,  4722
 len   EQU $-a2
 dst   DW len DUP(0)
.code
 _paddusw_ex proc
   mov     EAX, len
   shr     EAX, 1
   mov     EBX, 4
   xor     EDX, EDX
   div     EBX
   mov     ECX, EAX
   lea     ESI, a1
   lea     EDI, a2
   lea     EBX, dst
next:
   movq    MM0, qword ptr [ESI]
   paddusw MM0, qword ptr [EDI]
   movq    qword ptr [EBX], MM0
   add     ESI, 8
   add     EDI, 8
   add     EBX, 8
   dec     ECX
   jnz     next
   cmp     EDX, 0
   je      exit
   mov     ECX, EDX
next1:
   mov     AX, word ptr [ESI]
   add     AX, word ptr [EDI]
   mov     word ptr [EBX], AX
   add     ESI, 2
   add     EDI, 2
   add     EBX, 2
   dec     ECX
   jnz     next1
 exit:
   lea     EAX, dst
   ret
 _paddusw_ex endp
 end
```

Мы уже встречали подобный программный код, и я не буду на этом останавливаться. А вот результаты вычислений для этой процедуры мы проанализируем более подробно. Составим, как и для предыдущего примера, схему сложения:

```
al →      12094 31890 4107 41499 32054 35901 45033
+
a2 →      43701 39109 43771 29507 7199 2894 4722
-----------------------------------------------
al + a2 → 55795 65535 47878 65535 39253 38795 49755
```

Здесь жирным шрифтом выделены интересующие нас результаты. Из них видно, что если сумма беззнаковых чисел превышает верхнее значение для 16-разрядных беззнаковых чисел (65 535), то именно это значение и становится результатом.

Последний пример — сложение упакованных чисел со знаком, имеющих размер слова, при помощи команды paddsw. Результат сложения формируется по принципу знакового насыщения. Фрагменты программного кода, иллюстрирующие работу команды, показаны в листинге 12.7.

**Листинг 12.7.** Сложение упакованных чисел со знаком при помощи команды paddsw

```
.data
 al  DW 21609. -13104. -30011. -9081. 31209. 12056. 21305
 a2  DW 27791. -5959. -3290. 1544.  -4407. -32099. -7901
 len EQU $-a2
 res DW 7 dup (0)
.code
 . . .
 lea   ESI. al
 lea   EDI. a2
 lea   EBX. res
 . . .
 movq  MM0. qword ptr [ESI]
 movq  MM1. qword ptr [EDI]
 paddsw MM0. MM1
 movq  qword ptr [EBX]. MM0
 . . .
```

Результат сложения элементов массивов 16-разрядных чисел со знаком выглядит так:

```
al →      21609 -13104 -30011 -9081 31209 12056 21305
+
a2 →      27791 -5959 -3290 1544 -4407 -32099 -7901
-------------------------------------------------
al + a2 → 32767 -19063 -32768 -7537 26802 -20043 13404
```

Интересующие нас результаты выделены жирным шрифтом. Сумма первых элементов массивов al и a2 превышает максимально допустимое значение для 16-разрядных чисел со знаком (32 767), поэтому оно и берется в качестве результата. Сумма третьих элементов массивов меньше нижней границы диапазона допустимых значений, поэтому в качестве результата берется значение –32 768.

На этом рассмотрение MMX-команд сложения можно закончить и перейти к анализу команд вычитания.

# 12.3. Команды вычитания

В группу команд вычитания входят следующие MMX-команды:

- psub, psubw, psubd — вычитание элементов данных (байтов, слов или двойных слов) входного операнда из элементов данных выходного операнда. Если результат выходит за границу допустимого диапазона, то, по правилам циклической арифметики, соответствующее число единиц отсчитывается от другой границы диапазона. «Переноса» единицы из одного элемента данных в другой не происходит. Входной операнд может находиться в MMX-регистре или в памяти, а выходной операнд — в MMX-регистре;
- psubsb, psubsw — вычитание элементов данных (байтов или слов) входного операнда из элементов данных выходного операнда. Результат формируется по принципу знакового насыщения: если разность выходит за граничное значение допустимого диапазона, то результатом считается это граничное значение. Входной операнд может находиться в MMX-регистре или в памяти, а выходной — в MMX-регистре;
- psubusb, psubusw — вычитание элементов данных входного операнда из элементов данных выходного операнда. Результат формируется по принципу беззнакового насыщения: если разность выходит за граничное значение допустимого диапазона, то результатом считается это граничное значение. Входной операнд может находиться в MMX-регистре или в памяти; выходной операнд должен находиться в MMX-регистре.

Команды вычитания работают с теми же типами данных и формируют результат точно так же, как и только что рассмотренные команды сложения. Я не буду останавливаться подробно на анализе этих команд, думаю, читатель легко справится с этим сам. Приведу лишь пример демонстрационной процедуры (она называется _psubsw_ex), выполняющей вычитание 16-разрядных чисел со знаком и использующей команду psubsw (листинг 12.8).

**Листинг 12.8.** Вычитание чисел со знаком при помощи команды psubsw

```
.686
.model flat
.MMX
option casemap:none
.data
 a1 DW  1609, -13104, -30011, -9081,   -21209,  12056,  21305
 a2 DW  27791, -25959,  7290,   25544,  -9407,  -3099, -7901
 len EQU $-a2
 res DW len dup (0)
.code
 _psubsw_ex proc
   mov    EAX, len
   shr    EAX, 1
   mov    EBX, 4
   xor    EDX, EDX
   div    EBX
   mov    ECX, EAX
```

продолжение ⇗

**Листинг 12.8** (продолжение)

```
   lea    ESI, a1
   lea    EDI, a2
   lea    EBX, res
next:
   movq   MM0, qword ptr [ESI]
   movq   MM1, qword ptr [EDI]
   psubsw MM0, MM1
   movq   qword ptr [EBX], MM0
   emms
   add    ESI, 8
   add    EDI, 8
   add    EBX, 8
   dec    ECX
   jnz    next
   cmp    EDX, 0
   je     exit
   mov    ECX, EDX
next1:
   mov    AX, word ptr [ESI]
   sub    AX, word ptr [EDI]
   mov    word ptr [EBX], AX
   add    ESI, 2
   add    EDI, 2
   add    EBX, 2
   dec    ECX
   jnz    next1
exit:
   lea    EAX, word ptr res
   ret
 _psubsw_ex endp
 end
```

Процедура возвращает адрес результата (массив res, содержащий суммы чисел) в регистре EAX. Вызывающая эту процедуру программа на Visual C++ .NET выглядит так, как показано в листинге 12.9.

**Листинг 12.9.** Демонстрационная программа для процедуры из листинга 12.8

```
#include <stdio.h>
extern "C" short int* psubsw_ex(void);
int main(void)
{
  short int* psubsw = psubsw_ex();
  printf("PSUBSW result:\n");
  for (int i1 = 0;i1 < 7; i1++)
  {
   printf("%d ", *psubsw++);
  }
  return 0;
}
```

Поскольку процедура _psubsw_ex возвращает адрес массива 16-разрядных чисел со знаком, в вызывающей программе должен быть определен указатель типа short int*. Вызываемая процедура, как обычно, объявлена с директивой extern.

Результат работы процедуры можно схематически представить таким образом:

```
a1 →       1609 -13104 -30011 -9081 -21209 12056 21305
+
a2 →       27791 -25959 7290 25544 -9407 -3099 -7901
-------------------------------------------------------
a1 + a2 → -26182 12855 -32768 -32768 -11802 15155 29206
```

Поскольку команда вычитания знаковых слов psubsw формирует результат по принципу знакового насыщения, то две разности (выделенные жирным шрифтом) вышли за пределы нижней границы для 16-разрядных чисел со знаком и им было присвоено значение –32768.

Следующая, очень важная группа команд, которую мы рассмотрим, — это команды упаковки и распаковки данных.

## 12.4. Команды упаковки и распаковки данных

MMX-команды упаковки преобразуют длинные элементы данных (16- и 32-разрядные слова) в более короткие. Если исходное значение «не помещается» в коротком элементе данных, то происходит «насыщение» — результатом считается граничное значение допустимого диапазона выходного типа данных. Команды распаковки попарно объединяют элементы данных из обоих операндов в более длинные элементы выходного операнда. Этими командами можно пользоваться для увеличения числа значащих разрядов при вычислениях.

К этой группе команд относятся:

- packsswb, packssdw — преобразование длинных элементов данных (16- и 32-разрядных слов со знаком) в более короткие (байты или 16-разрядные слова со знаком). Если исходное значение было за пределами допустимого диапазона для выходного типа данных, то результатом упаковки считается ближайшее граничное значение диапазона. Входным операндом может выступать MMX-регистр или ячейка памяти, выходной операнд должен находиться в MMX-регистре;
- packuswb — выполняет преобразование 16-разрядных слов со знаком из обоих операндов в байты без знака и записывает их в выходной операнд. Если исходное слово со знаком было больше FFh, результатом преобразования считается FFh. Если исходное слово со знаком отрицательно, результатом преобразования считается 00h. Входным операндом может выступать MMX-регистр или ячейка памяти, выходной операнд должен находиться в MMX-регистре.

Работа команд упаковки станет более понятной, если посмотреть на рисунки, иллюстрирующие их работу. Например, следующая команда функционирует так, как показано на рис. 12.7:

```
packsswb MM0, MM1
```

Здесь операндом-источником является регистр MM1, а операндом-приемником — регистр MM0. Четыре слова регистра MM1 упаковываются в старшие четыре байта регистра-приемника MM0, а четыре слова регистра MM0 — в младшие четыре байта MM0.

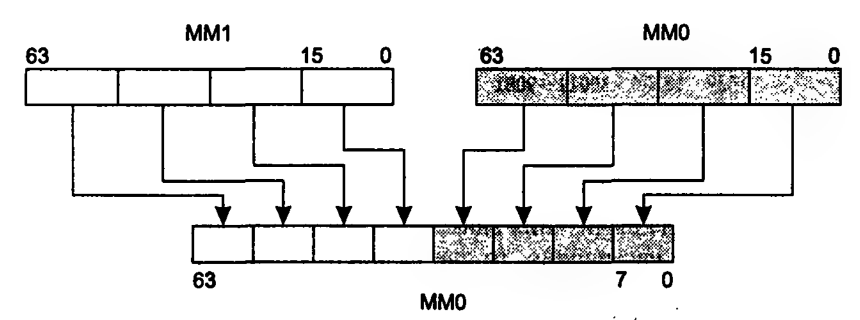



**Рис. 12.7.** Функционирование команды packsswb

Рисунок 12.8 иллюстрирует принцип работы команды

```
packssdw MM1, MM2
```

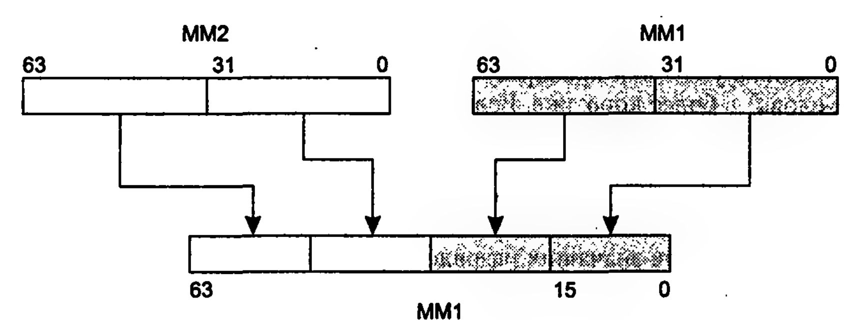



**Рис. 12.8.** Функционирование команды packssdw

В этом случае операндом-источником является регистр MM2, а операндом-приемником — регистр MM1. Два двойных слова регистра MM2 упаковываются в два старших слова регистра-приемника MM1, а два двойных слова регистра MM1 — в младшие два слова регистра MM1.

Рассмотрим практические примеры применения команд упаковки в программах на ассемблере. Первый пример реализован как процедура _packsswb_ex и демонстрирует работу команды packsswb (листинг 12.10).

**Листинг 12.10.** Упаковка знаковых слов в байты при помощи команды packsswb

```
.686
.model flat
.MMX
option casemap:none
.data
  a1  DW 45, -41, 67, -134, -61, 10, -88, 12, -62, 161,-99
  len EQU $-a1
  res DB len DUP(0)
.code
 _packsswb_ex proc
   mov  EAX, len
   shr  EAX, 1
   xor  EDX, EDX
```

```
  mov  EBX, 2
  div  EBX
  mov  ECX, EAX
  lea  ESI, a1
  lea  EDI, res
 next:
  movq     MM0, qword ptr [ESI]
  packsswb MM0, qword ptr [ESI+8]
  movq     qword ptr [EDI], MM0
  add      ESI, 16
  add      EDI, 8
  dec      ECX
  jnz      next
  cmp      EDX, 0
  je       exit
  mov      AL, byte ptr [ESI]
  mov      byte ptr [EDI], AL
 exit:
  lea      EAX, res
  ret
 _packsswb_ex endp
 end
```

Здесь элементы размером в слово из массива a1 упаковываются в байты массива res. Процедура _packsswb_ex возвращает адрес массива res в регистре EAX. Второй пример демонстрирует работу команды packssdw (листинг 12.11). Процедура _paxkssdw_ex выполняет упаковку двойных слов из массива a1 в элементы однословного массива res.

**Листинг 12.11.** Упаковка знаковых двойных слов в слова

```
.686
.model flat
.MMX
option casemap:none
.data
  a1  DD 7345,-4123,671,-34802,-611,75056,-8893,12,-6227,41161,-9991
  len EQU $-a1
  res DW len DUP(0)
.code
 _packssdw_ex proc
  mov  EAX, len
  shr  EAX, 2
  xor  EDX, EDX
  mov  EBX, 2
  div  EBX
  mov  ECX, EAX
  lea  ESI, a1
  lea  EDI, res
 next:
  movq     MM0, qword ptr [ESI]
  packssdw MM0, qword ptr [ESI+8]
  movq     qword ptr [EDI], MM0
  add      ESI, 16
```

продолжение ➩

**Листинг 12.11** (продолжение)

```
  add     EDI, 8
  dec     ECX
  jnz     next
  cmp     EDX, 0
  je      exit
  mov     AX, word ptr [ESI]
  mov     word ptr [EDI], AX
exit:
  lea     EAX, res
  ret
_packssdw_ex endp
end
```

Следующие команды этой процедуры приводят размер массива, выраженный в байтах, к количеству двойных слов:

```
mov  EAX, len
shr  EAX, 2
```

Для одновременной обработки 8-байтовых чисел необходимо выделить группы элементов по 2 двойных слова, что и выполняет следующий фрагмент кода:

```
xor  EDX, EDX
mov  EBX, 2
div  EBX
mov  ECX, EAX
```

После выполнения этих команд в регистре ECX окажется количество учетверенных слов, а регистр EDX будет содержать оставшиеся элементы массива, требующие отдельной обработки.

Основной цикл обработки:

```
next:
   movq     MM0, qword ptr [ESI]
   packssdw MM0, qword ptr [ESI+8]
   . . .
   je       exit
```

Этот цикл содержит команду, выполняющую упаковку четырех двойных слов (двух в регистре MM0 и двух из ячейки памяти с адресом [ESI+8]):

```
packssdw MM0, qword ptr [ESI+8]
```

Оставшийся одиночный элемент размером в двойное слово обрабатывается обычными командами:

```
mov     AX, word ptr [ESI]
mov     word ptr [EDI], AX
```

Последняя команда перед выходом из процедуры сохраняет адрес массива res в регистре EAX.

Третий пример более сложен по сравнению с предыдущими и демонстрирует сложение операндов размером в слово, предварительно упакованных в байты. В листинге 12.12 представлен исходный текст процедуры (она называется _add_pack_bytes), которая выполняет эти операции.

**Листинг 12.12.** Сложение слов, упакованных в байты

```
.686
.model flat
.MMX
option casemap:none
.data
  a1      DW 45, -41, 67, -134, -61, 10, -88, 12, -62, 61,-99
  b1      DW 37, -19, 122, 54,  88,   19, 133, 49, 13, 11,-29
  len     EQU $-b1
  a1_copy DB len DUP(0)
  b1_copy DB len DUP(0)
  res     DB len DUP(0)
.code
 _add_pack_bytes proc
   mov  EAX, len
   shr  EAX, 1
   xor  EDX, EDX
   mov  EBX, 2
   div  EBX
   mov  ECX, EAX
   push ECX
   push EDX
   lea  ESI, a1
   lea  EDI, a1_copy
   call convert_to_bytes
   lea  ESI, b1
   lea  EDI, b1_copy
   pop  EDX
   pop  ECX
   call convert_to_bytes
   mov  EAX, len
   mov  EBX, 8
   xor  EDX, EDX
   div  EBX
   mov  ECX, EAX
   lea  ESI, a1_copy
   lea  EDI, b1_copy
   lea  EBX, res
 again:
   movq MM0, qword ptr [ESI]
   paddsb MM0, qword ptr [EDI]
   movq   qword ptr [EBX], MM0
   add    ESI, 8
   add    EDI, 8
   add    EBX, 8
   dec    ECX
   jnz    again
   cmp    EDX, 0
   je     exit
   mov    ECX, EDX
next_byte:
   mov    AX, word ptr [ESI]
   add    AX, word ptr [EDI]
```

продолжение

**Листинг 12.12** (продолжение)

```
  mov    word ptr [EBX]. AX
  add    ESI. 2
  add    EDI. 2
  add    EBX. 2
  dec    ECX
  jnz    next_byte
exit:
  lea      EAX. res
  ret
  convert_to_bytes proc
next:
  movq     MM0. qword ptr [ESI]
  packsswb MM0. qword ptr [ESI+8]
  movq     qword ptr [EDI]. MM0
  add      ESI. 16
  add      EDI. 8
  dec      ECX
  jnz      next
  cmp      EDX. 0
  je       quit
  mov      ECX. EDX
next1:
  mov      AL. byte ptr [ESI]
  mov      byte ptr [EDI]. AL
  inc      ESI
  inc      EDI
  dec      ECX
  jnz      next1
quit:
  ret
convert_to_bytes endp
_add_pack_bytes endp
end
```

Анализ работы процедуры начнем с описания данных. В области данных процедуры определены пять массивов:

- a1 и b1 — исходные значения однословных чисел со знаком;
- a1_copy и b1_copy — упакованные в байты однословные элементы из массивов a1 и b1;
- res — суммы упакованных байтов.

Алгоритм работы процедуры _add_pack_bytes можно представить так: вначале размер массивов в байтах приводится к размерности в словах с помощью команд

```
mov  EAX. len
shr  EAX. 1
```

Затем из массивов a1 и b1 выделяются группы элементов по 4 слова (8 байт) и количество этих групп помещается счетчик ECX:

```
xor  EDX. EDX
mov  EBX. 2
div  EBX
mov  ECX. EAX
```

Количество элементов, оставшихся вне групп, помещается в регистр EDX.

Далее выполняются преобразование элементов массивов a1 и b1 из слов в байты и сохранение преобразованных элементов в массивах a1_copy и b1_copy:

```
lea  ESI, a1
lea  EDI, a1_copy
call convert_to_bytes
lea  ESI, b1
lea  EDI, b1_copy
pop  EDX
pop  ECX
call convert_to_bytes
```

Преобразование элементов обеспечивает подпрограмма convert_to_bytes. С помощью этой подпрограммы массивы слов a1 и b1 преобразуются в массивы байтов (a1_copy и b1_copy соответственно), после чего и выполняется сложение. Результаты сложения сохраняются в массиве res.

Подпрограмма convert_to_bytes для преобразования использует команду

```
packsswb MM0, qword ptr [ESI+8]
```

Эта команда повторяется в цикле, счетчик которого находится в регистре ECX и кратен восьми. После каждой итерации указатель адреса исходного массива (регистр ESI) перемещается на 16, а указатель массива-приемника (регистр EDI) — на 8. По завершении цикла next оставшиеся элементы массива просто копируются из массива a1 в b1.

После преобразования выполняется сложение упакованных байтов массивов a1_copy и b1_copy при помощи команды

```
paddsb MM0, qword ptr [EDI]
```

Это происходит в цикле

```
again:
 . . .
 paddsb MM0, qword ptr [EDI]
 . . .
 dec   ECX
 jnz   again
  . . .
```

Хочу напомнить, что при упаковке операндов необходимо учитывать диапазон формата упакованных чисел, поскольку легко потерять значимость результата!

Для отображения результатов сложения на экране используется программа, написанная на Visual C++ .NET (листинг 12.13).

**Листинг 12.13.** Демонстрационная программа для процедуры из листинга 12.12

```
#include <stdio.h>
extern "C" signed char* add_pack_bytes(void);
int main(void)
{
 signed char* add_pack = add_pack_bytes();
```

продолжение ⇗

**Листинг 12.13** (продолжение)

```
printf("SUMMA OF PACKED BYTES: \n");
for (int i1 = 0;i1 < 11;i1++)
 {
  printf("%d ", *add_pack++);
 }
return 0;
}
```

При указанных значениях элементов массивов результат на экране будет выглядеть так:

```
SUMMA OF PACKED BYTES:
82 -60 127 -74 27 29 39 61 -95 86 -128
```

Сейчас мы рассмотрим команды, выполняющие обратное действие по сравнению с командами упаковки — распаковку данных:

- punpckhbw, punpckhwd, punpckhdq — попарное объединение исходных элементов данных (байтов, 16- или 32-разрядных слов), находившихся в старших 32 разрядах обоих операндов. Полученные в результате более длинные элементы данных записываются в выходной операнд. Исходные значения младших разрядов операндов на результат не влияют. Входным операндом может выступать MMX-регистр или ячейка памяти, выходной операнд должен находиться в MMX-регистре;
- punpcklbw, punpcklwd, punpckldq — попарное объединение исходных элементов данных (байтов, 16- или 32-разрядных слов), находившихся в младших 32 разрядах обоих операндов. Полученные в результате более длинные элементы данных записываются в выходной операнд. Исходные значения старших разрядов операндов на результат не влияют. Входным операндом может выступать MMX-регистр или ячейка памяти, выходной операнд должен находиться в MMX-регистре.

Механизм работы команд распаковки станет более понятным, если посмотреть на рисунки, иллюстрирующие их работу. Например, следующая команда функционирует так, как показано на рис. 12.9:

```
punpckhbw MM0, MM1
```

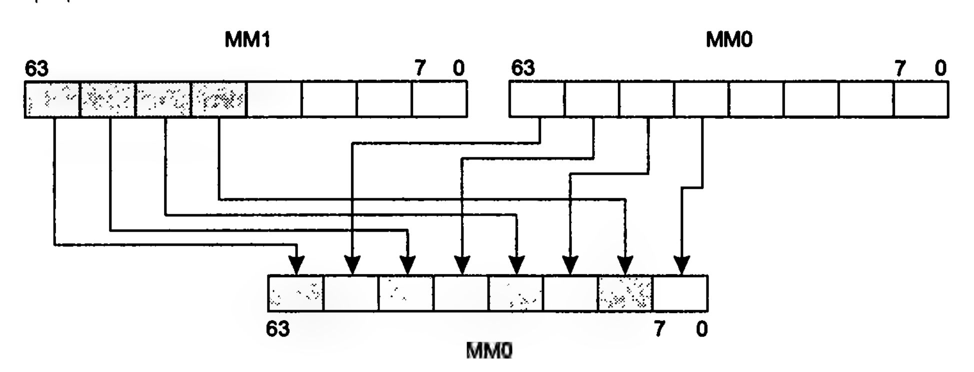



**Рис. 12.9.** Работа команды punpckhbw

Команда punpckhbw распаковывает старшие четыре байта операндов в операнд-приемник. Для рассматриваемой схемы операндом-источником является регистр MM1, а регистром-приемником — MM0. Как видно из рис. 12.9, старшие четыре байта регистра MM1 становятся старшими байтами однословных элементов регистра MM0, а старшие четыре байта регистра MM0 — младшими байтами однословных элементов.

Еще одна схема, показанная на рис. 12.10, иллюстрирует функционирование команды

```
punpckhwd MM2. MM3
```

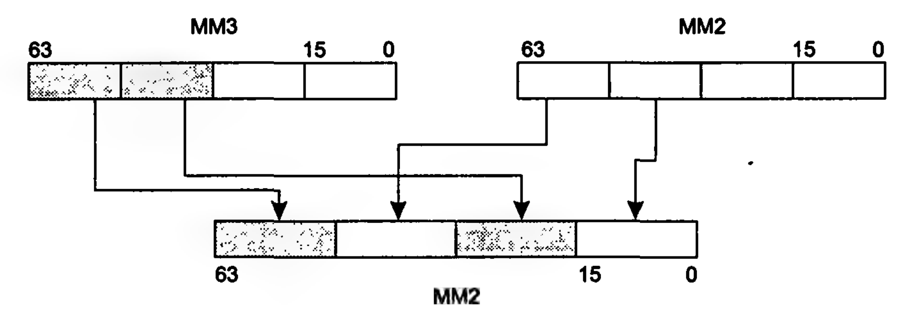



**Рис. 12.10.** Работа команды punpckhwd

Команда punpckhwd распаковывает старшие два слова операндов в операнд-приемник. Для рассматриваемой схемы операндом-источником является регистр MM2, а операндом-приемником — регистр MM2. Из рис. 12.10 видно, что старшие два слова регистра MM3 становятся старшими словами двухсловных элементов регистра MM2, а старшие два слова регистра MM2 — младшими словами двухсловных элементов в MM2.

Рисунок 12.11 иллюстрирует функционирование команды

```
punpckhdq MM0. MM1
```

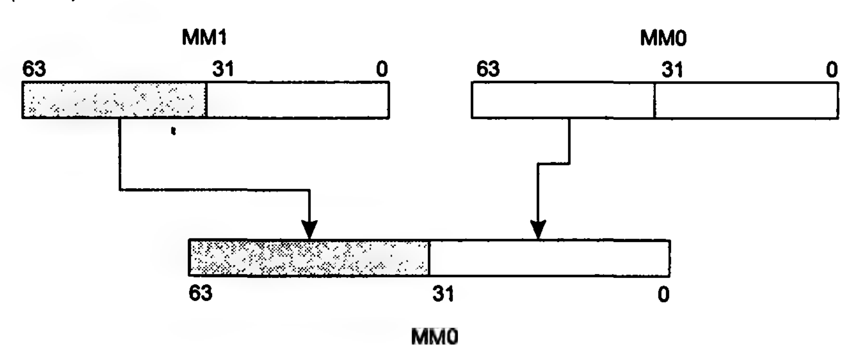



**Рис. 12.11.** Работа команды punpckhdq

Команда punpckhdq распаковывает старшие двойные слова операндов в операнд-приемник. Для рассматриваемой схемы операндом-источником является

регистр MM1, а операндом-приемником — регистр MM0. Из рис. 12.11 видно, что старшее двойное слово регистра MM1 становятся старшим двойным словом учетверенного слова в регистре MM0, а старшее двойное слово регистра MM0 — младшим двойным словом учетверенного слова в MM0.

Если команды punpckhbw и punpckhwd оперируют старшими частями 64-разрядных операндов, то команды punpcklbw, punpcklwd и punpckldq обрабатывают младшие части 64-разрядных операндов. Последующие схемы демонстрируют работу этих команд. На рис. 12.12 показана схема функционирования команды

```
punpcklbw MM0, MM1
```

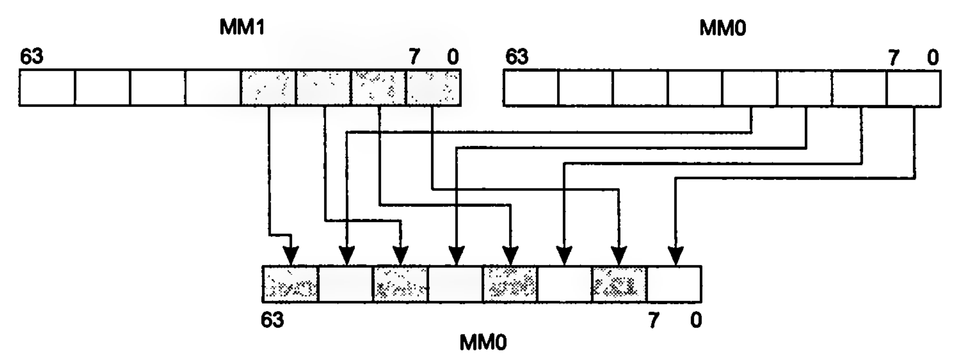



**Рис. 12.12.** Работа команды punpcklbw

На рис. 12.13 показана схема работы команды

```
punpcklwd MM1, MM2
```

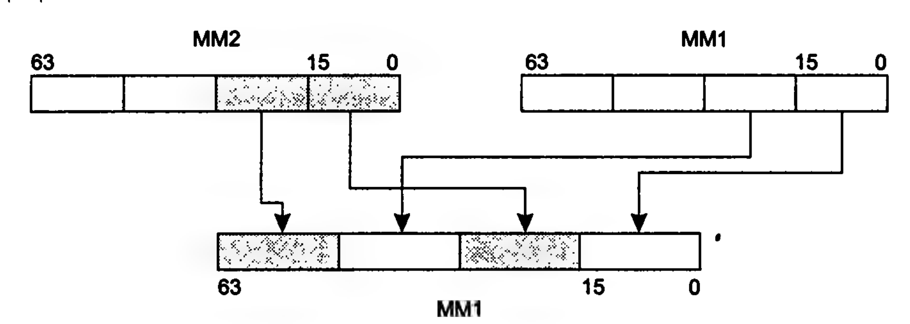



**Рис. 12.13.** Работа команды punpcklwd

На рис. 12.14 показана схема работы команды

```
punpckldq MM3, MM4
```

Рассмотрим практические примеры применения команд распаковки в программах на ассемблере. Первый пример демонстрирует работу команд punpckhbw и punpcklbw. Процедура называется _punpckbw_ex (листинг 12.14).

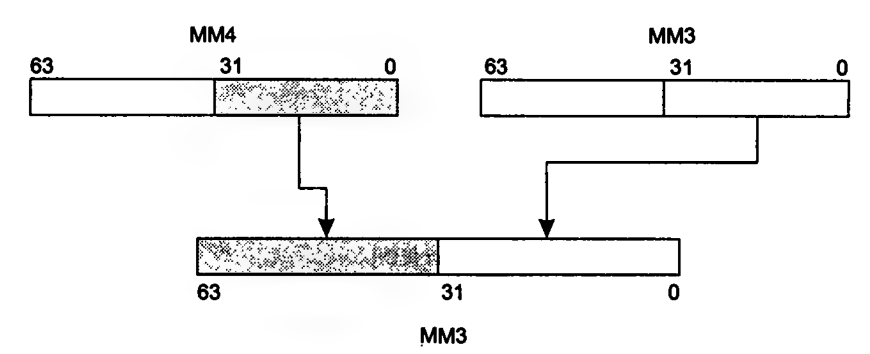



**Рис. 12.14.** Работа команды punpckldq

**Листинг 12.14.** Распаковка байтов в слова при помощи команд punpckhbw и punpcklbw

```
.686
.model flat
.MMX
option casemap: none
.data
 s1  DB '+++ ++++'
 s2  DB '32107654'
 res DB 16 DUP (' ').0
.code
 _punpckbw_ex proc
   lea       ESI, s1
   lea       EDI, s2
   lea       EBX, res
   movq      MM0, qword ptr [EDI]
   punpckhbw MM0, qword ptr [ESI]
   movq      qword ptr [EBX], MM0
   add       EBX, 8
   movq      MM0, qword ptr [EDI]
   punpcklbw MM0, qword ptr [ESI]
   movq      qword ptr [EBX], MM0
   lea       EAX, res
   ret
 _punpckbw_ex endp
 end
```

Алгоритм работы процедуры можно описать так: разместить символы строки s2 в порядке убывания и поместить между ними символы + строки s1. Эта последовательность символов должна размещаться в новой строке res. В этом случае результирующая строка res должна выглядеть как «7 + 6 + 5 + 4 + 3 + 2 + 1 + 0». Процедура возвращает в основную программу адрес этой строки в регистре EAX. Несмотря на относительно простой программный код, результат, вообще говоря, не очевиден, поэтому мы проведем детальный анализ работы процедуры _punpckbw_ex.

Для адресации всех трех строк используются регистры ESI, EDI и EBX, что и отражено в первых строках программного кода:

```
lea     ESI, s1
lea     EDI, s2
lea     EBX, res
```

Далее, в регистр MM0 с помощью следующей команды загружается строка s2:

```
movq MM0, qword ptr [EDI]
```

Теперь проанализируем содержимое регистра MM0, который является операндом-приемником, и содержимое строки s1, являющейся операндом-источником (рис. 12.15).

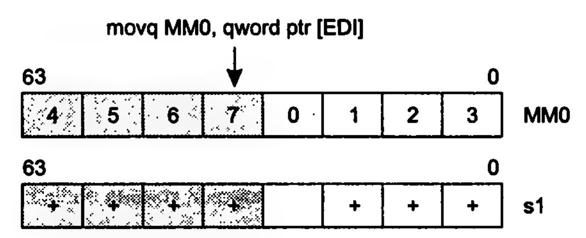



**Рис. 12.15.** Состояние операндов перед выполнением команды punpckhbw

На рисунке серым цветом выделены старшие 4 байта операндов. После выполнения следующей команды результат помещается в регистр MM0, а его содержимое формируется так, как показано на рис. 12.16:

```
punpckhbw MM0, qword ptr [ESI]
```

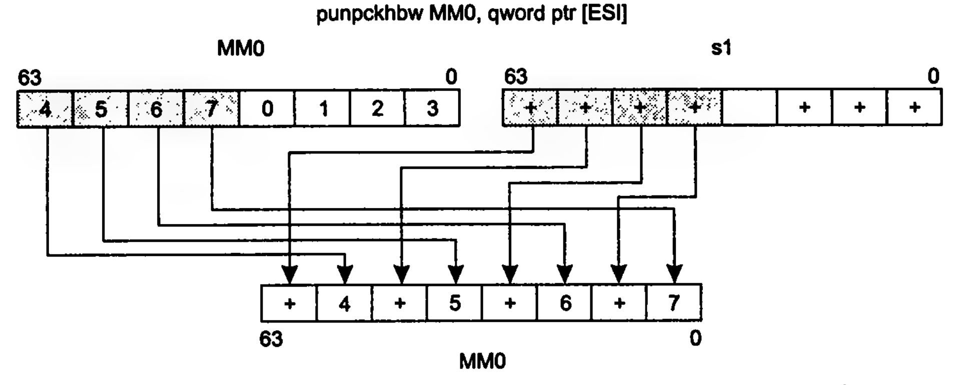



**Рис. 12.16.** Состояние операндов после выполнения команды punpckhbw

Затем результат, находящийся в регистре MM0, сохраняется в младших восьми байтах переменной res при помощи команды

```
movq qword ptr [EBX], MM0
```

Обратите внимание, что младший байт строки res представлен символом 7. Далее смещаем указатель результирующей строки res на 8 (команда add EBX, 8) и обрабатываем младшие 4 байта операндов с помощью команд

```
movq      MM0, qword ptr [EDI]
punpcklbw MM0, qword ptr [ESI]
movq      qword ptr [EBX], MM0
```

Среди этих команд особый интерес вызывает команда

```
punpcklbw MM0, qword ptr [ESI]
```

Результат ее выполнения схематически можно представить рис. 12.17 (серым цветом выделены обрабатываемые байты).

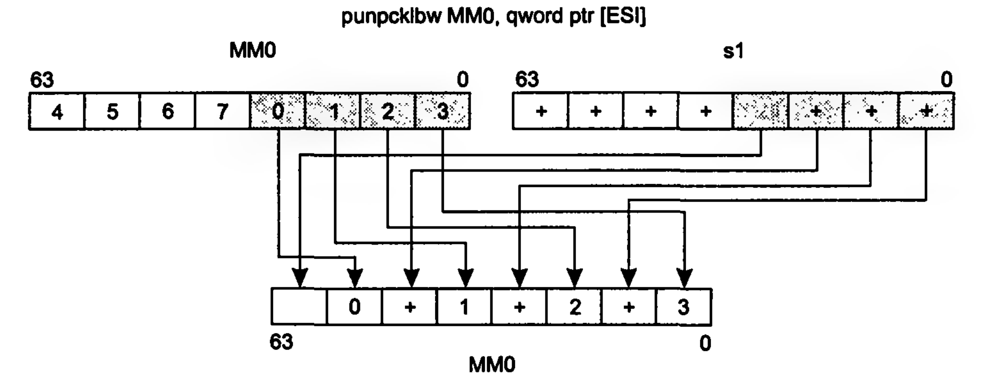



**Рис. 12.17.** Состояние операндов после выполнения команды punpcklbw

На этом обсуждение программного кода процедуры _punpckbw_ex можно закончить. Для проверки результата ее выполнения можно воспользоваться программой на Visual C++ .NET (листинг 12.15).

**Листинг 12.15.** Демонстрационная программа для процедуры из листинга 12.14

```
#include <stdio.h>
extern "C" char* punpckbw_ex(void);
int main(void)
{
 char* punpckbw = punpckbw_ex();
 printf("PUNPCKHBW/PUNPCKLBW example:\n");
 printf("%s\n", punpckbw);
   return 0;
}
```

Программа выводит на экран следующие строки:

```
PUNPCKHBW/PUNPCKLBW example:
7+6+5+4+3+2+1+0
```

Работу еще двух команд — punpckhwd и punpcklwd — демонстрирует следующий пример (листинг 12.16). Ассемблерная процедура _punpckwd_ex формирует в массиве res последовательность чисел, расположенных в двух массивах — a1 и a2 — в диапазоне 100–107.

**Листинг 12.16.** Распаковка слов в двойные слова при помощи команд punpckhwd и punpcklwd

```
.686
.model flat
.MMX
option casemap: none
.data
```

продолжение ⇗

**Листинг 12.16** (продолжение)

```
a1  label   qword
    DW 104. 106.100. 102
a2  label   qword
    DW 105. 107. 101. 103
res DD 4 DUP (0)
.code
_punpckwd_ex proc
  lea       ESI. a1
  lea       EDI. a2
  lea       EBX. res
  movq      MM0. qword ptr [ESI]
  punpckhwd MM0. qword ptr [EDI]
  movq      qword ptr [EBX]. MM0
  add       EBX. 8
  movq      MM0. qword ptr [ESI]
  punpcklwd MM0. qword ptr [EDI]
  movq      qword ptr [EBX]. MM0
  lea       EAX. res
  ret
_punpckwd_ex endp
end
```

В процедуре определены три массива целых чисел, содержащие двухбайтовые (однословные) элементы: a1, a2 и res. Элементы массивов a1 и a2 распаковываются в массив res, состоящий из четырех двойных слов. Как и в предыдущем примере, первые три команды lea загружают в регистры ESI, EDI и EBX адреса, но только не строк, а массивов. Далее в регистр MM0 помещаются элементы массива a1, после чего выполняется распаковка старших слов операнда-источника (массив a2, указатель в EDI) и операнда-приемника (регистр MM0) с помощью команды

```
punpckhwd MM0. qword ptr [EDI]
```

Эту операцию иллюстрирует схема, показанная на рис. 12.18 (задействованные в операции элементы показаны серым цветом).

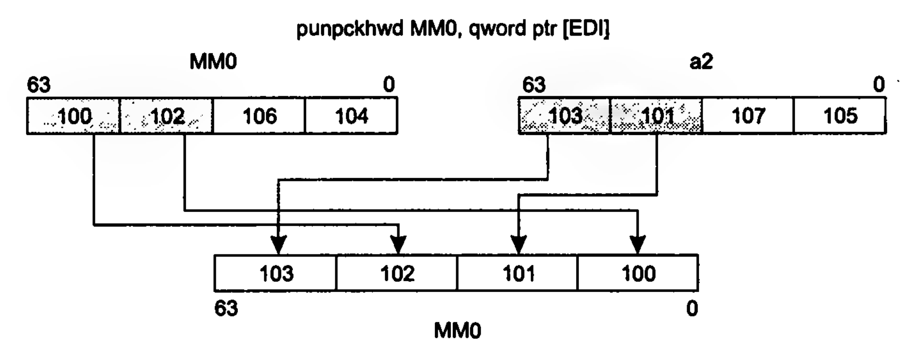



**Рис. 12.18.** Состояние операндов после выполнения команды punpckhwd

Как видно из рисунка, старшие слова операнда a2 помещаются в старшие слова двух двойных слов в регистре MM0, а старшие слова операнда-источника в MM0 — в младшие слова двойных слов в регистре MM0.

Далее указатель адреса для массива res продвигается на 8 (команда add EBX, 8), после чего выполняется распаковка младших слов массивов a1 и a2 в массив res. Ключевую роль в этом процессе играет команда

```
punpcklwd MM0, qword ptr [EDI]
```

Схема ее работы продемонстрирована на рис. 12.19.

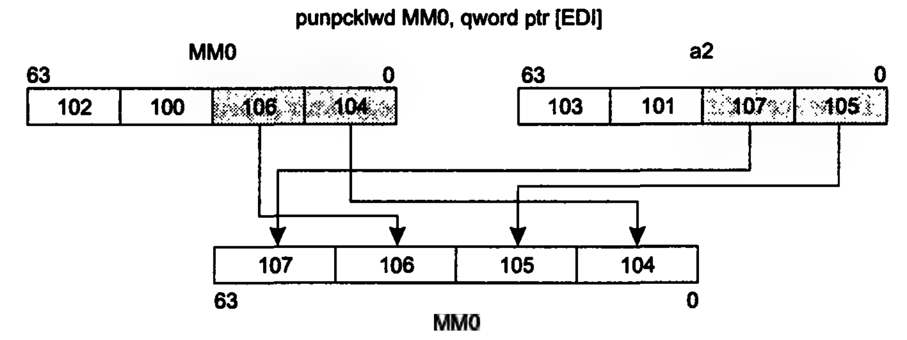



**Рис. 12.19.** Состояние операндов после выполнения команды punpcklwd

Младшие слова операнда a2 помещаются в старшие слова двух двойных слов в регистре MM0, а младшие слова операнда-источника в MM0 — в младшие слова двойных слов в регистре MM0. Обрабатываемые операнды выделены на рисунке серым цветом.

Для визуального отображения результатов работы процедуры служит программа на Visual C++ .NET (листинг 12.17).

**Листинг 12.17.** Демонстрационная программа для процедуры из листинга 12.16

```
#include <stdio.h>
extern "C" short int* punpckwd_ex(void);
int main(void)
{
 short int* punpckwd = punpckwd_ex();
 printf("PUNPCKHWD/PUNPCKLWD example:\n");
 for (int i1 = 0;i1 < 8;i1++)
  {
   printf("%d ", *punpckwd++);
  }
 return 0;
}
```

Программа вызывает процедуру punpckwd_ex, объявленную внешней (директива extern), и выводит следующий результат:

```
PUNPCKHWD/PUNPCKLWD example:
100 101 102 103 104 105 106 107
```

Перейдем к рассмотрению следующей группы MMX-команд — группы, в которую входят команды умножения.

## 12.5. Команды умножения

MMX-команды умножения попарно перемножают 16-разрядные слова операндов, что дает четыре 32-разрядных произведения. Все команды формируют результат по принципу циклической арифметики:

- pmaddwd — попарное умножение 16-разрядных слов со знаком, находящихся в двух операндах. После получения в результате четырех 32-разрядных произведений первое произведение складывается со вторым, а третье — с четвертым. Суммы записываются в 32-разрядные слова выходного операнда. Если все слова на входе равны 8000h, результат равен 80000000h (единственный случай, когда перемножение отрицательных чисел дает отрицательный результат). Входным операндом может выступать MMX-регистр или ячейка памяти, а выходным операндом должен быть MMX-регистр. Алгоритм работы команды pmaddwd показан на рис. 12.20;

pmaddwd MM0, MM1

| | | | | |
|---|---|---|---|---|
| x1 | x2 | x3 | x4 | MM0 |

+

| | | | | |
|---|---|---|---|---|
| x1 | x2 | x3 | x4 | MM1 |

=

| | | |
|---|---|---|
| x1 × y1 + x2 × y2 | x3 × y3 + x4 × y4 | MM0 |

**Рис. 12.20.** Умножение 16-разрядных слов с помощью команды pmaddwd

- pmulhw — попарное умножение 16-разрядных слов со знаком, находящихся во входном и выходном операндах. Результатом операции являются четыре 32-разрядных произведения, при этом старшие разряды произведений сохраняются в 16-разрядных словах выходного операнда, а младшие разряды произведений теряются. Входным операндом может выступать MMX-регистр или ячейка памяти, а выходным операндом должен быть MMX-регистр;
- pmullw — попарное умножение 16-разрядных слов со знаком входного и выходного операндов, что дает четыре 32-разрядных произведения. Младшие разряды произведений сохраняются в 16-разрядных словах выходного операнда, а старшие разряды произведений теряются. Входным операндом может выступать MMX-регистр или ячейка памяти, а выходным операндом должен быть MMX-регистр.

Команды pmulhw и pmullw позволяют выполнить умножение четырех 16-разрядных операндов одновременно, при этом разрядность результатов умножения оказывается в два раза больше разрядности операндов. Для получения полного результата умножения с помощью этих команд необходимо выполнить такую последовательность шагов:

1. Получить старшие 16 бит произведения, используя команду pmulhw.
2. Получить младшие 16 бит произведения, используя команду pmullw.

3. Объединить частичные результаты в одно двойное слово с помощью команд punpckhwd и punpcklwd.

Рассмотрим примеры использования команд умножения. Вначале проанализируем алгоритм работы процедуры _multiply_ex, в которой выполняются команды pmulhw и pmullw. Процедура обеспечивает умножение 16-разрядных целочисленных операндов из двух массивов, a1 и a2, и сохраняет 32-разрядные результаты в массиве res. Исходный текст процедуры представлен в листинге 12.18.

**Листинг 12.18.** Умножение 16-разрядных чисел двух массивов при помощи команд pmulhw и pmullw

```
.686
.model flat
.MMX
option casemap:none
.data
  a1  DW 34, -56, 29, 91, -5, 27, 139, 44, -791, -30, -802
  a2  DW -12, 3, -52, 23, -67, 322, -501, 122, -7, -15, 199
  len EQU $-a2
  res DD len DUP(0)
.code
 _multiply_ex proc
   mov   EAX, len
   shr   EAX, 1
   mov   EBX, 4
   xor   EDX, EDX
   div   EBX
   mov   ECX, EAX
   lea  ESI, a1
   lea  EDI, a2
   lea  EBX, res
next:
   movq   MM1, qword ptr [ESI]
   movq   MM0, qword ptr [EDI]
   pmulhw MM0, MM1
   movq   MM2, qword ptr [EDI]
   pmullw MM1, MM2
   movq   MM2, MM0
   movq   MM3, MM1
   punpckhwd MM3, MM2
   punpcklwd MM1, MM0
   movq   qword ptr [EBX], MM1
   movq   qword ptr [EBX+8], MM3
   add    ESI, B
   add    EDI, 8
   add    EBX, 16
   dec    ECX
   jnz    next
   cmp    EDX, 0
   jz     exit
   mov    ECX, EDX
next1:
   mov    AX, word ptr [ESI]
   imul   word ptr [EDI]
   mov    word ptr [EBX], AX
   mov    word ptr [EBX+2], DX
```

продолжение ⇗

**Листинг 12.18** (продолжение)

```
   add   ESI, 2
   add   EDI, 2
   add   EBX, 4
   dec   ECX
   jnz   next1
exit:
   lea   EAX, res
   ret
 _multiply_ex endp
 end
```

Наша процедура может использовать массивы 16-разрядных чисел произвольного размера. При этом для выполнения MMX-команд умножения удобно разбить элементы массивов на группы по 4 слова, а оставшиеся элементы обработать обычным образом. Для этого нужно установить в счетчики циклов соответствующие значения, что и выполняют команды

```
mov   EAX, len
shr   EAX, 1
mov   EBX, 4
xor   EDX, EDX
div   EBX
```

После выполнения этих команд регистр ECX будет содержать количество 8-байтовых групп элементов, а регистр EDX — количество оставшихся элементов.

Как обычно, перед началом цикла загружаем в регистры ESI, EDI и EBX адреса всех массивов с помощью команд lea. Далее начинается обработка элементов массива в цикле next. С помощью следующих команд вычисляются старшие слова 4-словных произведений элементов массивов a1 и a2:

```
movq   MM1, qword ptr [ESI]
movq   MM0, qword ptr [EDI]
pmulhw MM0, MM1
```

Результаты произведений помещаются в регистр MM0. Содержимое регистра MM1 после выполнения операции остается неизменным, чем мы воспользуемся в дальнейшем. Смысл произведенных действий иллюстрирует рис. 12.21, на котором показано вычисление старших слов произведений с помощью команды

```
pmulhw MM0, MM1
```

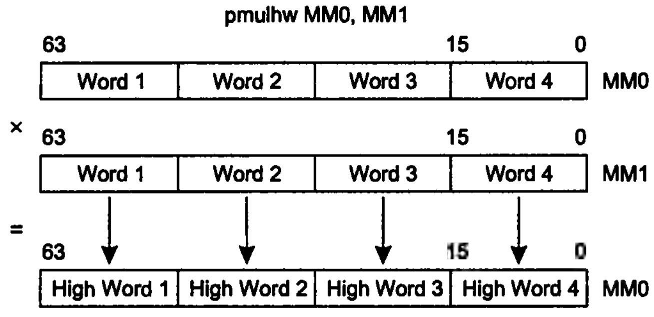



**Рис. 12.21.** Вычисление старших слов произведений командой pmulhw

Чтобы получить младшие слова произведения, выполняются команды

```
movq   MM2, qword ptr [EDI]
pmullw MM1, MM2
```

После выполнения этих команд результаты находятся в регистре MM1. Это иллюстрирует рис. 12.22.

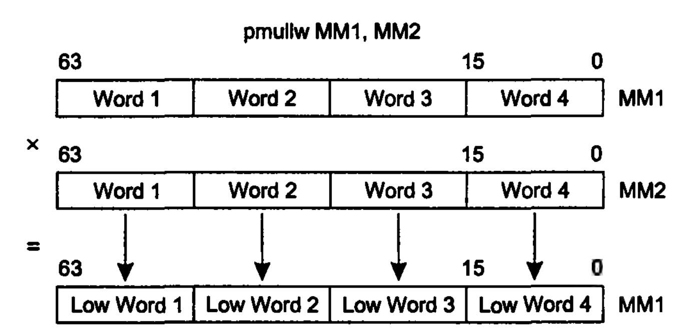



**Рис. 12.22.** Вычисление младших слов произведения командой pmullw

Таким образом, после попарного умножения четырех слов из массивов a1 и a2 старшие слова результатов находятся в регистре MM0, а младшие слова этого же произведения — в регистре MM1.

Следующий шаг, который нужно сделать, — объединить старшую и младшую части результатов и поместить полученные значения в массив res:

```
movq   MM2, MM0
movq   MM3, MM1
punpckhwd MM3, MM2
punpcklwd MM1, MM0
movq   qword ptr [EBX], MM1
movq   qword ptr [EBX+8], MM3
```

Алгоритм работы команд punpckhwd и punpcklwd мы уже рассматрели, смысл остальных команд достаточно очевиден, и я не буду на них останавливаться.

После выполнения всех итераций в цикле next оставшиеся элементы обрабатываются обычными командами в цикле next1. Хочу заметить, что для умножения 16-разрядных элементов в этом цикле используется команда imul (элементы массива считаются числами со знаком). Процедура возвращает результат в регистре EAX, в который помещается адрес массива res.

Тестирование процедуры можно выполнить с помощью программы на Visual C++ .NET (листинг 12.19).

**Листинг 12.19.** Демонстрационная программа для процедуры из листинга 12.18

```
#include <stdio.h>
extern "C" int* multiply_ex(void);
int main(void)
{
 int* pm = multiply_ex();
 printf("MULTIPLICATION example:\n");
 for(int i1 = 0; i1 < 11;i1++)
```

продолжение ⇒

**Листинг 12.19** (продолжение)

```
  {
   printf("%d ", *pm++);
  }
 return 0;
}
```

При заданных значениях элементов массивов программа выводит на экран следующие результаты:

```
MULTIPLICATION example:
-408 -168 -1508 2093 335 8694 -69639 5368 5537 450 -159598
```

В команде pmaddwd для реализации умножения используется несколько другой подход. Лучше всего продемонстрировать это на практике. Следующий пример показывает, как с помощью команды pmaddwd можно умножить две пары двойных чисел. Процедура называется _pmaddwd_ex, и ее исходный текст занимает всего несколько строк (листинг 12.20).

**Листинг 12.20.** Умножение двойных чисел при помощи команды pmaddwd

```
.686
.model flat
.MMX
option casemap: none
.data
 a1  DD 5893,-4592
 a2  DD 5275,5565
 res DQ 0
.code
 _pmaddwd_ex proc
   lea       ESI, a1
   lea       EDI, a2
   lea       EBX, res
   movq      MM0, qword ptr [ESI]
   pmaddwd   MM0, qword ptr [EDI]
   movq      qword ptr [EBX], MM0
   lea       EAX, res
   ret
 _pmaddwd_ex endp
 end
```

Как обычно, процедура возвращает результат в регистре EAX, куда помещается адрес массива res. Для вывода численных результатов можно воспользоваться программой, исходный текст которой представлен в листинге 12.21.

**Листинг 12.21.** Демонстрационная программа для процедуры из листинга 12.20

```
#include <stdio.h>
extern "C" int* pmaddwd_ex(void);
int main(void)
{
 int* pmaddwd = pmaddwd_ex();
 printf("PMADDWD:\n");
 printf("%d ", *pmaddwd++);
 printf("%d\n", *pmaddwd);
 return 0;
}
```

При указанных значениях элементов массивов a1 и a2 в процедуре _pmaddwd_ex программа выводит на экран значения 31 085 575 и –25 554 480.

На этом рассмотрение команд умножения можно закончить и перейти к анализу команд сравнения.

## 12.6. Команды сравнения

MMX-команды сравнения попарно сравнивают элементы данных (байты, 16- или 32-разрядные слова) входного и выходного операндов. В зависимости от результата сравнения соответствующий элемент данных выходного операнда заполняется нулями либо единицами. Эти команды, как и все остальные MMX-команды, не устанавливают флагов (признаков). В свою очередь, они делятся на две группы: команды обычного сравнения (равно или не равно) и команды сравнения по величине (больше или меньше). Операции сравнения проводятся для упакованных байтов, слов и двойных слов.

К командам сравнения относятся:

- pcmpeqb, pcmpeqw, pcmpeqd — попарное сравнение элементов данных (байтов, 16- или 32-разрядных слов) входного и выходного операндов. Если элемент данных выходного операнда равен соответствующему элементу входного, такой элемент выходного операнда заполняется единицами. Если элементы не равны, то он заполняется нулями. Входным операндом могут выступать MMX-регистр или ячейка памяти, а выходной операнд должен находиться в MMX-регистре;
- pcmpgtb, pcmpgtw, pcmpgtd — попарное сравнение элементов данных (байтов, 16- или 32-разрядных слов со знаком) входного и выходного операндов. Если элемент данных выходного операнда больше соответствующего элемента входного, такой элемент выходного операнда заполняется единицами, если же он не больше входного, то он заполняется нулями. Входной операнд может находиться в MMX-регистре или в памяти, а выходной операнд — в MMX-регистре.

Для операций обычного сравнения нулевые значения формируются, если соответствующие байты, слова или двойные слова не равны. Единичные значения формируются в случае, если соответствующие байты, слова или двойные слова равны. Для операций сравнения по величине единичные значения формируются, если соответствующие байты, слова или двойные слова первого операнда больше соответствующих байтов, слов или двойных слов второго операнда.

Лучше всего работу команд сравнения можно проиллюстрировать на примерах. В листинге 12.22 приводится исходный текст процедуры на ассемблере (она называется _pcmpeqd_ex), выполняющей сравнение по принципу «равно или не равно» двухсловных элементов целочисленных массивов.

Процедура принимает три входных параметра: два из них являются адресами сравниваемых массивов, а третий содержит размер массивов в байтах. Мнемонически процедуру _pcmpeqd_ex можно представить так:

```
pcmpeqd_ex(адрес_массива_1, адрес_массива_2, размер)
```

**Листинг 12.22.** Сравнение элементов целочисленных массивов на равенство/неравенство

```
.686
.model flat
.MMX
option casemap: none
.data
 res  DD 10 DUP (0)
.code
 _pcmpeqd_ex proc
   push    EBP
   mov     EBP, ESP
   mov     ECX, dword ptr [EBP+16]
   shr     ECX, 3
   mov     ESI, dword ptr [EBP+8]
   mov     EDI, dword ptr [EBP+12]
   lea     EBX, res
next:
   movq    MM0, qword ptr [ESI]
   pcmpeqd MM0, qword ptr [EDI]
   movq    qword ptr [EBX], MM0
   add     ESI, 8
   add     EDI, 8
   add     EBX, 8
   dec     ECX
   jnz     next
   lea     EAX, res
   pop     EBP
   ret
  _pcmpeqd_ex endp
  end
```

Для упрощения программного кода полагаем, что массивы имеют четное количество двойных слов и одинаковы по размеру.

Результат сравнения (единицы или нули в операнде-приемнике) помещается в массив результата res, адрес которого возвращается в вызывающую программу в регистре EAX.

Адреса массивов в регистры ESI, EDI и EBX загружают команды

```
mov     ESI, dword ptr [EBP+8]
mov     EDI, dword ptr [EBP+12]
lea     EBX, res
```

Регистр ECX после выполнения следующих команд содержит количество пар двойных слов и является счетчиком цикла, в котором будет выполняться сравнение элементов массивов:

```
mov     ECX, dword ptr [EBP+16]
shr     ECX, 3
```

Операция сравнения осуществляется с использованием команды

```
pcmpeqd MM0, qword ptr [EDI]
```

Эта команда выполняется в каждой итерации цикла:

```
next:
   movq    MM0, qword ptr [ESI]
   pcmpeqd MM0, qword ptr [EDI]
```

```
movq    qword ptr [EBX], MM0
. . .
jnz     next
```

Протестировать процедуру можно с помощью приложения на Visual C++ .NET (листинг 12.23).

**Листинг 12.23.** Демонстрационная программа для процедуры из листинга 12.22

```
#include <stdio.h>
extern "C" int* pcmpeqd_ex(int* a1, int* a2, int counter);
int main(void)
{
 int a1[] = { 345, 87, -455, 12, -5439, 99};
 int a2[] = { 345, 87, -455, 12, -5439, 98};
 int counter = sizeof(a2);
 bool flag = true;
 int* pq = pcmpeqd_ex(a1, a2, counter);
 for (int i1 = 0; i1 < 6; i1 ++)
  {
   if (*pq == 0xFFFFFFFF)
   {
    *pq++;
   }
   else
   {
    flag = false;
    break;
   }
  }
if (flag)
  printf("OPERANDS ARE EQUAL!\n");
else
  printf("OPERANDS ARE NOT EQUAL!\n");
return 0;
}
```

Поскольку последние элементы массивов a1 и a2 различаются, то результатом будет выведенная на экран строка:

```
OPERANDS ARE NOT EQUAL!
```

Результат сравнения последних пар двойных слов в процедуре _pcmpeqd_ex показан на рис. 12.23.

pcmpeqd MM0, qword ptr [EDI]

MM0

| | | |
|---|---|---|
| 99 | -5439 | a1 |

=?

| | | |
|---|---|---|
| 98 | -5439 | a2 |

=

| | |
|---|---|
| 00000000h | FFFFFFFFh |

**Рис. 12.23.** Результат сравнения последних пар элементов

Если в рассмотренном только что примере определялось лишь равенство/неравенство элементов, то следующий пример демонстрирует соотношение между операндами, исходя из их значений. В этом примере демонстрируется использование команды pcmpgtd. В листинге 12.24 представлен исходный текст процедуры _pcmpgtd_ex, выполняющей сравнение элементов массивов a1 и a2.

**Листинг 12.24.** Сравнение элементов массивов по их значениям

```
.686
.model flat
.MMX
option casemap: none
.data
 res  DD 10 DUP (0)
.code
 _pcmpgtd_ex proc
   push    EBP
   mov     EBP, ESP
   mov     ECX, dword ptr [EBP+16]
   shr     ECX, 3
   mov     ESI, dword ptr [EBP+8]
   mov     EDI, dword ptr [EBP+12]
   lea     EBX, res
next:
   movq    MM0, qword ptr [ESI]
   pcmpgtd MM0, qword ptr [EDI]
   movq    qword ptr [EBX], MM0
   add     ESI, 8
   add     EDI, 8
   add     EBX, 8
   dec     ECX
   jnz     next
   lea     EAX, res
   pop     EBP
   ret
  _pcmpgtd_ex endp
  end
```

Исходный текст данной процедуры похож на текст предыдущей, за исключением того, что используется другая команда сравнения (в тексте процедуры она выделена жирным шрифтом):

```
pcmpgtd MM0, dword ptr [EDI]
```

Хочу сделать одно важное замечание: команды pcmpgt работают со знаковыми операндами. Вызывающая программа на Visual C++ .NET отображает содержимое результирующего массива res (в шестнадцатеричном формате), куда помещаются двойные слова, содержащие нули или единицы, в зависимости от результата сравнения (листинг 12.25).

**Листинг 12.25.** Демонстрационная программа для процедуры из листинга 12.24

```
#include <stdio.h>
extern "C" int* pcmpgtd_ex(int* a1, int* a2, int counter);
int main(void)
```

```
{
 int a1[] = { 345, 87, -479, 12, -5459, 100};
 int a2[] = { 345, 87, -480, 12, -5469, 99};
 int counter = sizeof(a2);
 printf("PCMPGTD example:\n");
 int* pgt = pcmpgtd_ex(a1, a2, counter);
 for(int i1 = 0; i1 < 6; i1++)
 {
  printf("%Xh ", *pgt++);
 }
 return 0;
}
```

При указанных значениях элементов массивов программа выводит на экран следующий результат:

```
PCMPGTD example:
0h 0h FFFFFFFFh 0h FFFFFFFFh FFFFFFFFh
```

Такой результат является закономерным, поскольку 2, 4 и 5-й элементы массива a1 больше соответствующих элементов массива a2.

# 12.7. Логические команды

Логические MMX-команды выполняют поразрядные логические операции над всеми 64 битами своих операндов. Они реализуют логические операции И, ИЛИ, И-НЕ, исключающего ИЛИ:

- pand (логическое И) — вычисляет поразрядное логическое И своих операндов. Входной операнд может быть MMX-регистром или операндом в памяти. Выходной операнд должен находиться в MMX-регистре;
- pandn (логическое И-НЕ) — вычисляет обращение (поразрядное НЕ) выходного операнда, а затем поразрядное логическое И между входным операндом и обращенным значением выходного. Входным операндом могут выступать MMX-регистр или ячейка памяти. Выходной операнд должен находиться в MMX-регистре;
- por (логическое ИЛИ) — вычисляет поразрядное логическое ИЛИ своих операндов. Входной операнд может находиться в MMX-регистре или в ячейке памяти. Выходной операнд должен быть MMX-регистром;
- pxor (исключающее ИЛИ) — вычисляет поразрядное логическое исключающее ИЛИ своих операндов. Входной операнд может содержаться в MMX-регистре или в ячейке памяти. Выходной операнд должен находиться в MMX-регистре.

Работа команд достаточно очевидна, поэтому перейдем к примерам. В листинге 12.26 показан исходный текст процедур, выполняющих логические операции над двумя операндами, являющимися входными параметрами для этих процедур, причем все процедуры возвращают адрес массива в регистре EAX.

**Листинг 12.26.** Применение логических команд

```
.686
.model flat
option casemap:none
.MMX
.data
  res DQ 0
.code
 _pand_ex proc
   push EBP
   mov  EBP, ESP
   movq MM0, qword ptr [EBP+8]
   pand MM0, qword ptr [EBP+16]
   movq qword ptr res, MM0
   lea  EAX, res
   pop  EBP
   ret
 _pand_ex endp
 _pandn_ex proc
   push  EBP
   mov   EBP, ESP
   movq  MM0, qword ptr [EBP+8]
   pandn MM0, qword ptr [EBP+16]
   movq  qword ptr res, MM0
   lea   EAX, res
   pop   EBP
   ret
 _pandn_ex endp
 _por_ex proc
   push EBP
   mov  EBP, ESP
   movq MM0, qword ptr [EBP+8]
   por  MM0, qword ptr [EBP+16]
   movq qword ptr res, MM0
   lea  EAX, res
   pop  EBP
   ret
 _por_ex endp
 _pxor_ex proc
   push EBP
   mov  EBP, ESP
   movq MM0, qword ptr [EBP+8]
   pxor MM0, qword ptr [EBP+16]
   movq qword ptr res, MM0
   lea  EAX, res
   pop  EBP
   ret
 _pxor_ex endp
 end
```

Каждая из процедур (_pand_ex, _pandn_ex, _por_ex, _pxor_ex) получает через регистр EBP два 64-разрядных значения, после чего выполняет над ними соответствующие операции при помощи команд pand, pandn, por и pxor.

Для проверки работоспособности процедур можно использовать программный код, написанный на Visual C++ .NET (листинг 12.27).

**Листинг 12.27.** Демонстрационная программа для процедур из листинга 12.26

```
#include <stdio.h>
extern "C"  long long* pand_ex(long long a1, long long b1);
extern "C" long long* pandn_ex(long long a1, long long b1);
extern "C" long long* por_ex(long long a1, long long b1);
extern "C" long long* pxor_ex(long long a1, long long b1);
int main(void)
{
 long long a1 = 0x7AE1;
 long long b1 = 0xCD80;
 printf("PAND: %X\n", *pand_ex(a1, b1));
 printf("PANDN: %X\n", *pandn_ex(a1, b1));
 printf("POR: %X\n", *por_ex(a1, b1));
 printf("PXOR: %X\n", *pxor_ex(a1, b1));
 return 0;
}
```

Обратите внимание на то, что все параметры и возвращаемые значения в этой программе объявлены как long long — это объявление для 64-разрядных величин в Visual C++ .NET. Все процедуры объявлены как внешние директивой extern.

При указанных значениях переменных a1 и b1 (выделены жирным шрифтом) программа выводит на экран следующие результаты:

```
PAND: 4B80
PANDN: 8500
POR: FFE1
PXOR: B761
```

# 12.8. Команды сдвига

MMX-команды сдвига выполняют сдвиг каждого элемента данных (16-, 32- или 64-разрядного слова) в выходном операнде на величину, задаваемую входным операндом. Среди команд сдвига выделяют команды арифметического и логического сдвига. При выполнении команд арифметического сдвига освобождающиеся разряды элементов заполняются знаком числа (старший бит) и могут принимать значение как 0, так и 1, в то время как при выполнении команд логического сдвига освободившиеся разряды заполняются нулями. К этой группе относятся следующие команды:

- psllw, pslld, psllq — сдвиг элементов данных (16-, 32- или 64-разрядных слов) в выходном операнде на число битов, задаваемое входным операндом. Освободившиеся младшие разряды заполняются нулями. Входной операнд может быть непосредственным операндом либо находиться в MMX-регистре или в памяти. Выходной операнд должен находиться в MMX-регистре;
- psrlw, psrld, psrlq — сдвиг элементов данных (16-, 32- или 64-разрядных слов) в выходном операнде на число битов, задаваемое входным операндом. Освободившиеся старшие разряды заполняются нулями. Входной операнд может быть непосредственным операндом либо находиться в MMX-регистре или в памяти. Выходной операнд должен находиться в MMX-регистре;

- psraw, psrad — сдвиг элементов данных (16- или 32-разрядных слов) в выходном операнде на число битов, задаваемое входным операндом. Если сдвигается положительное число, то освободившиеся старшие разряды заполняются нулями, если отрицательное, то единицами. Входной операнд может быть непосредственным операндом либо находиться в MMX-регистре или в памяти. Выходной операнд должен находиться в MMX-регистре.

Работа этих команд иллюстрируется программным кодом процедур из листинга 12.28.

**Листинг 12.28.** Применение команд сдвига

```
.686
.model flat
option casemap:none
.MMX
.data
  res DQ 0
.code
 _psllq_ex proc
   push  EBP
   mov   EBP, ESP
   movq  MM0, qword ptr [EBP+8]
   psllq MM0, qword ptr [EBP+16]
   movq  qword ptr res, MM0
   lea   EAX, res
   pop   EBP
   ret
 _psllq_ex endp
 _psrlq_ex proc
   push  EBP
   mov   EBP, ESP
   movq  MM0, qword ptr [EBP+8]
   psrlq MM0, qword ptr [EBP+16]
   movq  qword ptr res, MM0
   lea   EAX, res
   pop   EBP
   ret
 _psrlq_ex endp
 _psrad_ex proc
   push  EBP
   mov   EBP, ESP
   movq  MM0, qword ptr [EBP+8]
   psrad MM0, qword ptr [EBP+16]
   movq  qword ptr res, MM0
   lea   EAX, res
   pop   EBP
   ret
 _psrad_ex endp
 end
```

В листинге представлен исходный текст процедур, демонстрирующих особенности операций логического сдвига двойных слов и арифметического сдвига слов. Процедура _psllq_ex показывает работу команды psllq, процедура _psrlq_ex — команды psrlq и, наконец, процедура _psrad_ex — команды psrad. Все процедуры в качестве параметров принимают исходное значение операнда по адресу [EBP+8]

и величину сдвига ([EBP+16]). Процедуры возвращают результат обычным образом, через регистр EAX, в который помещается адрес переменной res с полученным значением.

Визуальное отображение полученных результатов позволяет получить простая программа на Visual C++ .NET (листинг 12.29).

**Листинг 12.29.** Демонстрационная программа для процедуры из листинга 12.28

```
#include <stdio.h>
extern "C" long long* psllq_ex(long long a1, long long shft);
extern "C" long long* psrlq_ex(long long a1, long long shft);
extern "C" long long* psrad_ex(long long a1, long long shft);
int main(void)
{
 long long a1 = 0x9FFFFFFF;
 long long shft = 2;
 printf("PSLLQ: %X\n", *psllq_ex(a1, shft));
 printf("PSRLQ: %X\n", *psrlq_ex(a1, shft));
 printf("PSRAD: %X\n", *psrad_ex(a1, shft));
 return 0;
}
```

Программа выводит содержимое 64-разрядных значений (спецификатор long long) в шестнадцатеричном формате (спецификатор X). Для указанного значения переменной a1 (0x9FFFFFFF) результаты работы программы будут выглядеть так:

```
PSLLQ: 7FFFFFFCh
PSRLQ: 27FFFFFFh
PSRAD: E7FFFFFFh
```

Полученные результаты легко интерпретируются с помощью схемы на рис. 12.24. Здесь старшая часть 64-разрядных операндов изображена в уменьшенном виде, а биты младшей части представлены полностью.

Работа команды psllq MM0, 2 при MM0 = 9FFFFFFFh

Содержимое регистра MM0 до операции
1001 1111 1111 1111 1111 1111 1111 1111
31 0

Содержимое регистра MM0 после операции
0010 0111 1111 1111 1111 1111 1111 1111
31 0

Работа команды psrlq MM0, 2 при MM0 = 9FFFFFFFh

Содержимое регистра MM0 до операции
1001 1111 1111 1111 1111 1111 1111 1111
31 0

Содержимое регистра MM0 после операции
0010 0111 1111 1111 1111 1111 1111 1111
31 0

Работа команды psrad MM0, 2 при MM0 = 9FFFFFFFh

Содержимое регистра MM0 до операции
1001 1111 1111 1111 1111 1111 1111 1111
31 0

Содержимое регистра MM0 после операции
1110 0111 1111 1111 1111 1111 1111 1111
31 0

**Рис. 12.24.** Функционирование команд сдвига

## 12.9. Дополнительные команды

Сейчас мы рассмотрим еще одну группу команд, которые трудно отнести к какому-либо определенному типу, но которые являются весьма полезными при разработке программ. Эти команды включены во все поколения процессоров Intel Pentium, начиная с Pentium III. Вот некоторые из них:

- pavgb, pavdw — вычисляют среднее значение двух чисел, представленных байтами или словами. Значения операндов интерпретируются как беззнаковые целые числа. В качестве входного операнда могут выступать MMX-регистр или 64-разрядная ячейка памяти, выходным операндом служит один из MMX-регистров;
- pextrw — извлекает одно из четырех упакованных слов входного операнда. Команда имеет три аргумента: входной операнд, выходной операнд и маска. Младшие два бита маски задают во входном операнде номер упакованного слова, подлежащего извлечению. Извлеченное слово сохраняется в младшем слове выходного операнда. Выходным операндом этой команды может выступать один из 32-разрядных регистров общего назначения. Старшее слово выходного операнда обнуляется;
- pinsrw — вставляет слово в одно из четырех упакованных слов выходного операнда. Выходным операндом является один из MMX-регистров, а входным операндом может выступать один из 32-разрядных регистров общего назначения, младшее слово которого будет вставлено в выходной операнд. Номер позиции, куда помещается операнд, определяется младшими двумя битами маски и может принимать значения от 0 до 3;
- pmaxub, pmaxsw — извлекают максимальное значение из каждой пары упакованных элементов в выходном и входном операндах. Операции могут выполняться как над беззнаковыми байтами (pmaxub), так и над знаковыми словами (pmaxsw). В качестве выходного операнда может выступать MMX-регистр, а в качестве входного — MMX-регистр или 64-разрядная ячейка памяти;
- pminub, pminsw — извлекают минимальное значение из каждой пары упакованных элементов в выходном и входном операндах. Операции могут выполняться как над беззнаковыми байтами (pminub), так и над знаковыми словами (pminsw). В качестве выходного операнда может выступать MMX-регистр, а в качестве входного — MMX-регистр или 64-разрядная ячейка памяти;
- pmovmskb — формирует байт, содержащий знаковые биты восьми байтов, содержащихся во входном операнде, в качестве которого может выступать один из MMX-регистров. Выходным операндом является 32-разрядный регистр общего назначения, младший байт которого будет содержать результат. Эта команда очень удобна для формирования условных ветвлений в программах;
- psadbw — вычисляет суммарную разность значений беззнаковых байтов входного и выходного операндов. Входным операндом могут выступать MMX-регистр или 64-разрядная ячейка памяти, а выходным — один из MMX-регистров.

Рассмотрим пример использования команд pmaxsw. Здесь вычисляется максимальное значение из пар 16-разрядных чисел со знаком, содержащихся в массивах a1 и b1. Процедура (она называется pmaxsw_ex) на ассемблере выглядит так, как показано в листинге 12.30.

**Листинг 12.30.** Вычисление максимальных значений пар целых чисел со знаком

```
.686
.model flat
.MMX
option casemap:none
.data
  a1  DW 45, -67, 23, 11
  b1  DW -671, 223, 3, 155
  res DQ 0
.code
 pmaxsw_ex proc
   movq   MM0, qword ptr a1
   pmaxsw MM0, qword ptr b1
   movq   qword ptr res, MM0
   lea    EAX, res
   ret
 pmaxsw_ex endp
 end
```

Исходный текст процедуры несложен, и нет необходимости его анализировать.

На этом обзор возможностей команд MMX-расширения можно закончить.

Я надеюсь, что материал главы окажет помощь программистам в разработке сложных и эффективных программ.

# SSE-расширение процессоров Intel Pentium




В этой главе рассматриваются вопросы использования SSE-расширения при разработке приложений на ассемблере. Это расширение, впервые появившееся в процессорах Intel Pentium III, дополняет MMX-расширение средствами обработки данных с плавающей точкой.

SSE-расширение реализовано в виде аппаратно-программного модуля, который включает дополнительные восемь регистров разрядностью в 128 бит, имеющих обозначение `XMM0` – `XMM7`, и 32-разрядный регистр управления/состояния `MXCSR` (рис. 13.1).

127 0
XMM0
XMM1
XMM2
XMM3
XMM4
XMM5
XMM6
XMM7
31 0
MXCSR

**Рис. 13.1.** Аппаратная модель SSE-расширения

Программная часть SSE-расширения включает в себя набор SSE-команд для работы с данными в формате плавающей точки. Содержимое XMM-регистра представляет собой четыре 32-разрядных операнда с плавающей точкой в коротком формате (Single Precision Floating Point, SPFP). Представление данных SSE-расширения соответствует стандарту IEEE-754, а сам формат данных показан на рис. 13.2.

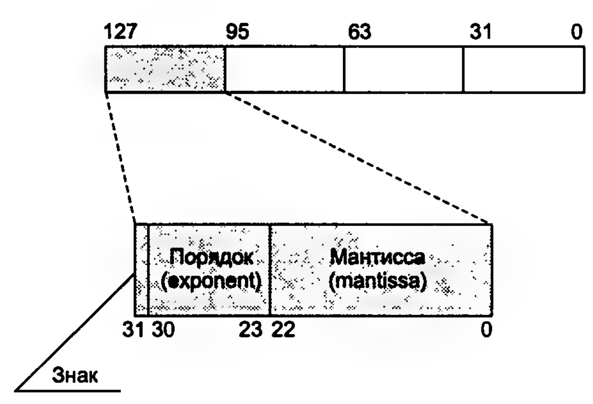



**Рис. 13.2.** Формат данных SSE-расширения

Здесь мантисса (mantissa) и порядок (exponent) формируют число в формате SPFP в соответствии с формулой

$$\text{мантисса} \times 2^{\text{порядок}}$$

Диапазон изменения чисел, представленных в данном формате, равен $2^{-126}$–$2^{127}$.

Следует отметить, что данный формат данных несопоставим с тем, который принят для математического сопроцессора (число в 80-разрядном расширенном формате), поэтому в некоторых случаях при разных границах выравнивания результаты вычислений с использованием форматов FPU и SSE могут различаться.

Поскольку аппаратно модуль SSE-расширения реализован независимо от других модулей, то это позволяет выполнять SSE-команды параллельно с командами математического сопроцессора и MMX-командами. При этом для синхронизации вычислений инструкции наподобие emms не требуются.

Структура полей регистра управления/состояния (MXCSR) во многом напоминает ту, что реализована в регистрах состояния (swr) и управления (cwr) математического сопроцессора. Состоянием вычислений можно управлять путем установки определенных значений в поля этого регистра.

Набор инструкций SSE-расширения включает 70 команд. Подробное описание всех SSE-команд — это тема отдельной книги, поэтому я приведу описание и примеры использования только части из них. Для более глубокого изучения архитектуры SSE и практического применения команд можно воспользоваться фирменным руководством Intel по процессорам Pentium III и выше.

Значительная часть команд может выполняться в двух контекстах: скалярном и параллельном. Это относится к арифметическим командам, а также к командам сравнения. Команды параллельных арифметических операций обрабатывают одновременно 4 двойных слова и имеют в своей мнемонике суффикс ps (рис. 13.3).

Команды скалярных операций обрабатывают только младшие 32-разрядные двойные слова упакованных операндов, не затрагивая при этом три старших двойных слова. Исключение составляют некоторые команды скалярной пересылки данных. Мнемоническое обозначение этих команд включает суффикс ss, схема их выполнения показана на рис. 13.4.




**Рис. 13.3.** Схема выполнения команд параллельных операций

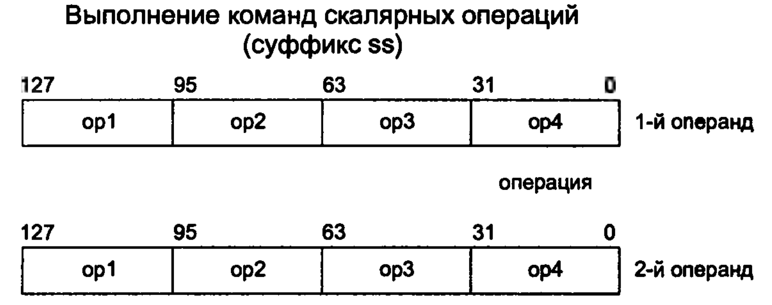



**Рис. 13.4.** Схема выполнения команд скалярных операций

Рассмотрим важнейшие группы команд SSE-расширения и методику их использования. Напомню, что для тестирования программного кода, представленного в этой главе (как и предыдущей), понадобится компилятор ассемблера фирмы Microsoft версии 7.10*xxxx* или любой другой, поддерживающий SSE-команды.

В процессе обработки данных команды SSE-расширения могут возбуждать исключительные ситуации, которые возникают, если происходит одно из следующих событий:

- некорректная операция (invalid operation);
- денормализованный операнд (denormalized operand);
- деление на 0 (divide-by-zero);
- арифметическое переполнение (numeric overflow);
- потеря значащих разрядов (numeric underflow);
- потеря точности (inexact result).

При возникновении исключительных ситуаций устанавливаются биты 0–5 в регистре управления/состояния (MXCSR). Каждая исключительная ситуация может быть замаскирована путем установки в 1 битов 7–12 регистра MXCSR. Если какое-либо исключение замаскировано, то оно обрабатывается процессором по стандартному алгоритму, после чего продолжается выполнение программного кода. Формат полей регистра MXCSR и их назначение показаны на рис. 13.5.

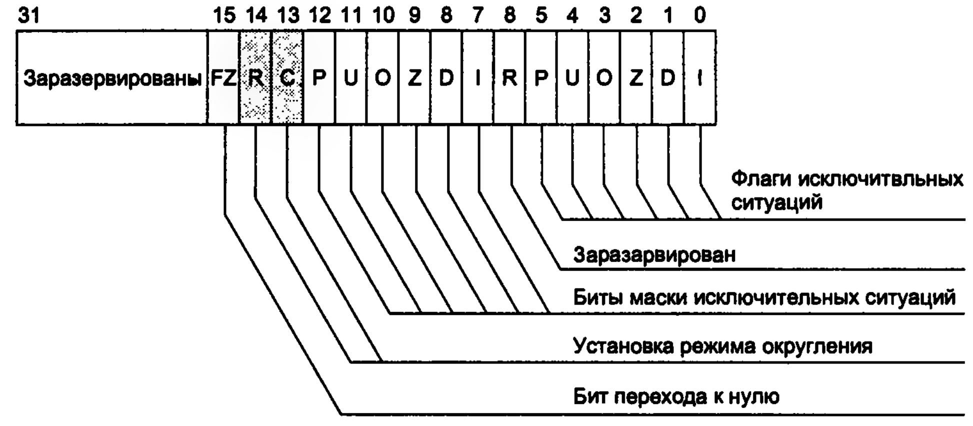



**Рис. 13.5.** Регистр управления/состояния

Биты 13–14 регистра MXCSR поля **RC** (или **rc**, что одно и то же) задают режим округления. По умолчанию устанавливается режим округления к ближайшему значению числа с плавающей точкой в коротком формате. Эти биты можно установить программно, причем возможны следующие комбинации:

- 00 — округление к ближайшему числу;
- 01 — округление к меньшему числу;
- 10 — округление к большему числу;
- 11 — округление с отбрасыванием дробной части.

Бит 15 используется, если результат операции близок к нулю. При этом процессор выполняет следующие действия:

- возвращает значение 0 и знак результата;
- устанавливает флаги P (бит 5) и U (бит 4);
- маскирует биты исключений.

Программная реализация SSE-расширения включает несколько десятков команд. Все их условно можно разделить на несколько групп:

- команды передачи данных;
- арифметические команды;
- команды сравнения;
- команды преобразования;
- логические команды;
- дополнительные команды.

Прежде чем приступить к анализу SSE-команд, следует определить, поддерживает ли данный процессор SSE-расширение. Это необходимо для того, чтобы узнать, можно ли использовать команды SSE-расширения при разработке

программ. Такую проверку можно выполнить при помощи простой процедуры (она называется _ssesupport), код которой представлен в листинге 13.1.

**Листинг 13.1.** Тест процессора на поддержку SSE

```
.686
.model flat
option casemap: none
.data
  supSSE DB 1
.code
_ssesupport proc
  mov  EAX, 1
  cpuid
  test EDX, 2000000h
  jnz  exit
  mov  supSSE, 0
exit:
  xor  EAX, EAX
  mov  AL, supSSE
  ret
_ssesupport endp
end
```

Процедура возвращает в регистре AL единицу, если процессор поддерживает SSE-расширение, и 0, если не поддерживает.

Приступим к анализу отдельных групп команд и начнем с команд передачи (пересылки) данных.

# 13.1. Команды передачи данных

В группу команд передачи данных входит несколько команд, позволяющих выполнять операции над 128-, 64- и 32-разрядными операндами:

- movaps — пересылка выровненных по 16-байтовой границе 128-разрядных операндов. В качестве входного операнда могут выступать XMM-регистр или 128-разрядная ячейка памяти, выходным операндом может служить один из XMM-регистров. Если адрес ячейки памяти не будет выровнен по 16-байтовой границе, то произойдет исключение общей защиты.
- movups — пересылка невыровненных данных; в качестве входного операнда могут выступать XMM-регистр или 128-разрядная ячейка памяти, выходным операндом может служить один из XMM-регистров. Команда не требует выравнивания по 16-байтовой границе.
- movhps — пересылка невыровненных 64 бит из входного операнда в выходной. Один из операндов обязательно должен быть XMM-регистром, в качестве второго может выступать 64-разрядная ячейка памяти. Пересылаются только старшие 64 бит входных операндов. Младшие 64 бит обоих операндов не изменяются. Если данные передаются из XMM-регистра, то пересылке подлежат только старшие 64 бит. Команда не требует выравнивания по 16-байтовой границе адреса ячейки памяти. Схема выполнения команды movhps показана на рис. 13.6.

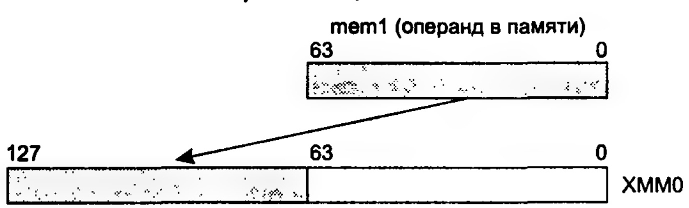



**Рис. 13.6.** Схема работы команды movhps

- movhlps — пересылка невыровненных 64 бит из входного операнда в выходной. Оба операнда должны находиться в XMM-регистрах. Пересылаются только старшие 64 бит входных операндов. В результате выполнения этой команды изменяются младшие 64 бит регистра-приемника. Схема выполнения команды movhlps показана на рис. 13.7.

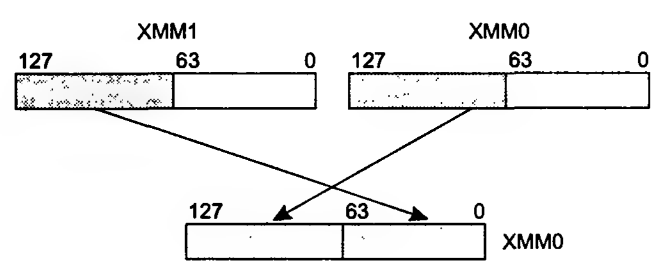



**Рис. 13.7.** Схема работы команды movhlps

- movlhps — пересылка невыровненных 64 бит из входного операнда в выходной. Оба операнда должны находиться в XMM-регистрах. Пересылаются только младшие 64 бит входных операндов. В результате выполнения этой команды изменяются старшие 64 бит регистра-приемника. Схема выполнения команды movlhps показана на рис. 13.8.

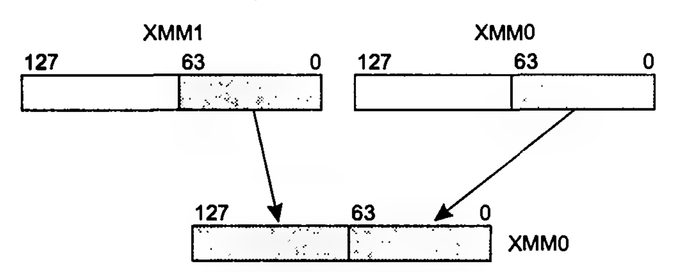



**Рис. 13.8.** Схема работы команды movlhps

- movlps — пересылка невыровненных 64 бит из входного операнда в выходной. Один из операндов обязательно должен быть XMM-регистром, в качестве

второго может выступать 64-разрядная ячейка памяти. Пересылаются только младшие 64 бит входных операндов. Старшие 64 бит обоих операндов не изменяются. Если данные передаются из XMM-регистра, то пересылке подлежат только младшие 64 бит. Команда не требует выравнивания по 16-байтовой границе адреса ячейки памяти. Схема выполнения команды movlps показана на рис. 13.9.

movlps XMM0, mem1

mem1 (операнд в памяти)

63 0

127 63 0

XMM0

**Рис. 13.9.** Схема работы команды movlps

- movmskps — пересылка знакового бита каждого из четырех упакованных чисел входного операнда в младшие четыре бита выходного операнда. В качестве входного операнда может выступать только XMM-регистр, а в качестве выходного операнда — 32-разрядный регистр общего назначения. Эта команда используется для организации условных переходов. Схема выполнения команды movmskps показана на рис. 13.10.

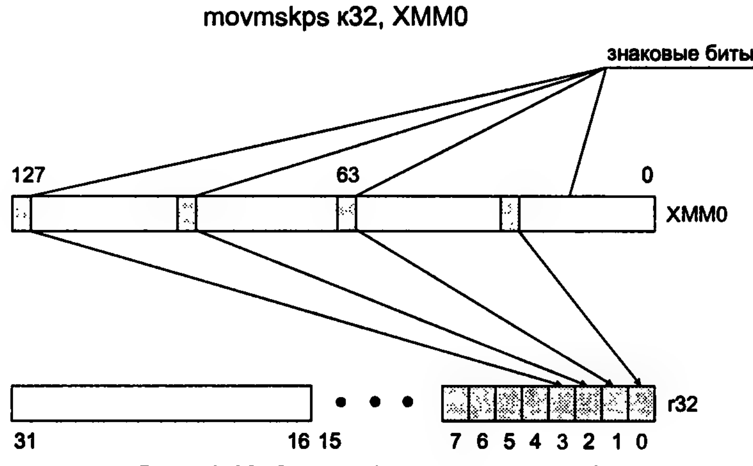



**Рис. 13.10.** Схема работы команды movmskps

- movss — пересылка 32 младших битов из источника в приемник, при этом как минимум один из операндов должен быть XMM-регистром. Второй операнд должен быть 32-разрядной ячейкой памяти. При выполнении операции пересылки из операнда в памяти младшие 32 бит помещаются в младшие 32 бит XMM-регистра. Если выполняется пересылка из XMM-регистра, то выбираются младшие 32 разряда регистра, остальные разряды не изменяются.

После описания команд пересылки данных рассмотрим несколько примеров их практического применения, но перед этим сделаю одно важное замечание — в исходном тексте ассемблерных программ, содержащих SSE-команды, обязательно должна присутствовать директива .XMM. В первом примере демонстрируется работа команд movhlps и movlhps. Программный код представлен двумя процедурами: _movhlps_ex и _movlhps_ex (листинг 13.2).

**Листинг 13.2.** Применение команд передачи данных

```
.686
.model flat
.XMM
option casemap:none
.data
 al  DD 0,2,4,6
 bl  DD 1,3,5,7
 len DD $-bl
 res DD 4 DUP(0)
.code
 _movhlps_ex proc
   movups  XMM0, al
   movups  XMM1, bl
   movhlps XMM0, XMM1
   movups  res, XMM0
   lea     EAX, res
   ret
 _movhlps_ex endp
 _movlhps_ex proc
   movups  XMM0, al
   movups  XMM1, bl
   movlhps XMM0, XMM1
   movups  res, XMM0
   lea     EAX, res
   ret
 _movlhps_ex endp
 end
```

В первой процедуре, _movhlps_ex, в XMM-регистры XMM0 и XMM1 помещаются элементы массивов al и bl соответственно. Это выполняется при помощи команд

```
movups  XMM0, al
movups  XMM1, bl
```

После этих операций XMM0 и XMM1 будут содержать элементы массивов в таком порядке, как показано на рис. 13.11.

| 127 — 96 | 95 — 64 | 63 — 32 | 31 — 0 | |
|---|---|---|---|---|
| 6 | 4 | 2 | 0 | XMM0 |

| 127 — 96 | 95 — 64 | 63 — 32 | 31 — 0 | |
|---|---|---|---|---|
| 7 | 5 | 3 | 1 | XMM1 |

**Рис. 13.11.** Содержимое регистров после выполнения команд movups

После выполнения следующей команды регистр XMM0 будет содержать значения двойных слов, показанные на рис. 13.12:

```
movhlps XMM0. XMM1
```

| 127 — 96 | 95 — 64 | 63 — 32 | 31 — 0 | |
|---|---|---|---|---|
| 6 | 4 | 7 | 5 | XMM0 |

**Рис. 13.12.** Содержимое регистров после выполнения команды movhlps

И наконец, команда

```
movups res. XMM0
```

После выполнения этой команды переменная res будет содержать последовательность данных, показанную на рис. 13.13.

Младший адрес → Старший адрес

| | | | | |
|---|---|---|---|---|
| 5 | 7 | 4 | 6 | res |

**Рис. 13.13.** Содержимое массива res после выполнения команды movups

Думаю, что читатель сможет самостоятельно проанализировать программный код процедуры _movlhps_ex.

Для тестирования только что рассмотренных ассемблерных процедур можно воспользоваться программным кодом приложения на Visual C++ .NET (листинг 13.3).

**Листинг 13.3.** Демонстрационная программа для процедур из листинга 13.2

```
#include <stdio.h>
extern "C" int* movhlps_ex(void):
extern "C" int* movlhps_ex(void):
int main(void)
{
 printf("MOVHLPS example:\n"):
 int* pi1 = movhlps_ex():
 for (int i1 = 0:i1 < 4: i1++)
  {
   printf("%d ". *pi1++):
  }
 printf("\nMOVLHPS example:\n"):
 int* pi2 = movlhps_ex():
 for (int i1 = 0:i1 < 4: i1++)
 {
  printf("%d ". *pi2++):
 }
return 0:
}
```

Результат, выведенный на экран:

```
MOVHLPS example:
5 7 4 6
MOVLHPS example:
0 2 1 3
```

Команда movmskps, как было отмечено, сохраняет в 32-разрядном регистре знаковые биты 4 упакованных чисел из XMM-регистра. Пример демонстрационной процедуры, анализирующей эти биты (она называется _movmskps_ex), приведен в листинге 13.4. В зависимости от значения знакового бита процедура возвращает в регистре EAX адрес строки, содержащей соответствующее сообщение о знаке числа. Программный код процедуры обрабатывает одновременно 4 числа из массива.

Процедура принимает три параметра (слева направо):

- адрес массива чисел с плавающей точкой;
- смещение в массиве — неотрицательное целое число;
- смещение элемента в группе (0–3).

Мнемонически процедуру можно представить так:

_movmskps_ex(*адрес_массива*, *смещение_в_массиве*, *смещение_элемента*)

**Листинг 13.4.** Применение команды movmskps

```
.686
.model flat
.XMM
option casemap:none
.data
 pos_number DB "Number is positive!", 0
 neg_number DB "Number is negative!", 0
.code
 _movmskps_ex proc   push      EBP
   mov       EBP, ESP
   mov       ESI, dword ptr [EBP+8]
   mov       EDX, dword ptr [EBP+12]
   shl       EDX, 2
   add       ESI, EDX
   mov       ECX, dword ptr [EBP+16]
   movups    XMM0, [ESI]
   movmskps  EBX, XMM0
   bt        EBX, ECX
   jc        neg_res
   lea       EAX, pos_number
   jmp       exit
 neg_res:
   lea       EAX, neg_number
 exit:
   pop       EBP
   ret
 _movmskps_ex endp
 end
```

Проведем краткий анализ исходного текста. Адрес массива помещается в регистр ESI командой

```
mov ESI, dword ptr [EBP+8]
```

Смещение группы из 4 элементов, требующих обработки, относительно начала массива вычисляется при помощи команд

```
mov  EDX, dword ptr [EBP+12]
shl  EDX, 2
```

Найденное смещение помещается в регистр EDX. Обратите внимание на то, что смещение пересчитывается в двойные слова, поскольку мы имеем дело с 32-разрядными числами. Затем устанавливаем указатель адреса на первый элемент группы, выполнив команду

```
add ESI, EDX
```

Теперь нужно получить смещение элемента массива в группе. Для этого помещаем значение третьего параметра в регистр ECX:

```
mov ECX, dword ptr [EBP+16]
```

Следующие две команды загружают в регистр XMM0 четыре двойных слова по смещению, указанному в регистре ESI, и помещают знаковые биты в регистр EBX:

```
movups    XMM0, [ESI]
movmskps  EBX, XMM0
```

Для лучшего понимания выполняемой операции представим, что есть массив вещественных чисел, например:

3,45, –6, –9,4, 2, –99, 34, 45, –1,22, 49, –25, –3,09

Предположим, нас интересуют знаки 5-го и 7-го элементов (это числа 34 и –1,22, нумерация начинается с нуля). Для определения знаков элементов можно вызвать процедуру с разными параметрами:

- для 5-го элемента:

  `_movmskps_ex(адрес_массива, 4, 1)`

- для 7-го элемента:

  `_movmskps_ex(адрес_массива, 4, 3)`

Вернемся к нашей программе. К этому моменту в битах 0–3 регистра EBX находятся знаковые биты 4 упакованных чисел из XMM0. Для того чтобы определить знак требуемого элемента, воспользуемся командой

```
bt EBX, ECX
```

Эта команда помещает значение бита, указанного в регистре ECX, во флаг переноса CF.

Далее содержимое флага переноса анализируется командой jc и, в зависимости от результата, в регистр EAX помещается адрес строки с сответствующим сообщением, после чего происходит выход из процедуры.

Перейдем к следующей группе команд, которая включает в себя арифметические команды сложения, вычитания, умножения и деления.

## 13.2. Арифметические команды

Группа арифметических команд включает команды сложения и вычитания, умножения и деления, также к ним часто относят и команды извлечения квадратного корня и вычисления максимального/минимального значения. Приступим к более детальному анализу арифметических команд и начнем с команд сложения:

- addps — параллельное сложение 128-разрядных операндов, при этом в качестве входного операнда (операнда-источника) выступают один из XMM-регистров или 128-разрядная ячейка памяти. Выходным операндом команды (операндом-приемником) является один из XMM-регистров;
- addss — скалярное сложение операндов; младшие двойные слова операндов должны быть числами с плавающей точкой в коротком формате (single precision). Результат помещается в операнд-приемник, в качестве которого может выступать XMM-регистр. Входным операндом может быть XMM-регистр или 32-разрядная ячейка памяти.

Следующий пример, реализованный в виде процедуры _addps_ex, демонстрирует применение команды addps (листинг 13.5).

**Листинг 13.5.** Параллельное сложение 128-разрядных операндов

```
686
.model flat
.XMM
option casemap:none
.data
 a1  DD 34.78, -56.07, -129.31, 94.2
 b1  DD -59.16, 44.93, -73.12, 19.61
 len EQU $-b1
 res DD  len DUP(0)
.code
 _addps_ex proc
  lea    ESI, a1
  lea    EDI, b1
  movups XMM0, [ESI]
  addps  XMM0, [EDI]
  movups res, XMM0
  lea    EAX, res
  ret
_addps_ex endp
 end
```

Сложение двух массивов, a1 и b1, можно выполнить и с помощью команды скалярного сложения addss, работающей с 32-разрядными операндами. Эта операция реализована в листинге 13.6 с помощью процедуры _addss_ex.

**Листинг 13.6.** Скалярное сложение элементов массивов

```
.686
.model flat
.XMM
option casemap:none
.data
 a1  DD 34.78, -56.07, -129.31, 94.2
 b1  DD -59.16, 44.93, -73.12, 19.61
 len EQU $-b1
 res DD  len DUP(0)
.code
 _addss_ex proc
  lea    ESI, a1
  lea    EDI, b1
  lea    EBX, res
```

продолжение ↗

**Листинг 13.6** (продолжение)

```
  mov    ECX, len
  shr    ECX, 2
next:
  movd   XMM0, dword ptr [ESI]
  addss  XMM0, dword ptr [EDI]
  movd   dword ptr [EBX], XMM0
  add    ESI, 4
  add    EDI, 4
  add    EBX, 4
  dec    ECX
  jnz    next
  lea    EAX, res
  ret
_addss_ex endp
 end
```

Сложение элементов массивов выполняется в цикле next:

```
next:
  movd   XMM0, dword ptr [ESI]
  addss  XMM0, dword ptr [EDI]
  movd   dword ptr [EBX], XMM0
  . . .
  jnz    next
```

В каждой итерации цикла складываются два 32-разрядных операнда. Для этого служит команда

```
addss XMM0, dword ptr [EDI]
```

К командам вычитания относятся следующие инструкции ассемблера:

- subps — параллельное вычитание 128-разрядных операндов; в качестве операнда-источника выступают один из XMM-регистров или 128-разрядная ячейка памяти. Выходным операндом команды является один из XMM-регистров;
- subss — скалярное вычитание операндов; младшие двойные слова операндов должны быть числами с плавающей точкой в коротком формате. Результат помещается в операнд-приемник, в качестве которого может выступать XMM-регистр. Входным операндом могут быть XMM-регистр или 32-разрядная ячейка памяти.

В листинге 13.7 показано использование команды subps для попарного вычитания элементов массивов a1 и b1 (процедура _subps_ex).

**Листинг 13.7.** Параллельное вычитание 128-разрядных операндов

```
.686
.model flat
.XMM
option casemap:none
.data
 a1  DD 34.78, -56.07, -129.31, 94.2
 b1  DD -59.16, 44.93, -73.12, 19.61
 len EQU $-b1
 res DD  len DUP(0)
.code
```

```
 _subps_ex proc
  lea    ESI, a1
  lea    EDI, b1
  movups XMM0, [ESI]
  subps  XMM0, [EDI]
  movups res, XMM0
  lea    EAX, res
  ret
_subps_ex endp
 end
```

Исходный текст процедуры _subps_ex во многом напоминает программный код ранее рассмотренной процедуры для параллельного сложения _addps_ex, за исключением того, что здесь используется команда subps вместо addps, поэтому останавливаться подробно на этом коде нет смысла.

Для ознакомления с работой команды скалярного вычитания subss можно воспользоваться исходным текстом процедуры _addss_ex, заменив в ней команду addss командой subss.

Теперь проанализируем работу команд параллельного и скалярного умножения:

- mulps — параллельное умножение 128-разрядных операндов; в качестве операнда-источника выступают один из XMM-регистров или 128-разрядная ячейка памяти. Выходным операндом команды должен быть один из XMM-регистров;
- mulss — скалярное умножение операндов; младшие двойные слова операндов должны быть числами с плавающей точкой в коротком формате. Результат помещается в операнд-приемник, которым должен быть XMM-регистр. Входным операндом могут быть XMM-регистр или 32-разрядная ячейка памяти.

Листинг 13.8 демонстрирует применение команды mulps, реализованной в виде процедуры _mulps_ex.

**Листинг 13.8.** Параллельное умножение 128-разрядных операндов

```
.686
.model flat
.XMM
option casemap:none
.data
 a1  DD 34.78, -56.07, -129.31, 94.2
 b1  DD -59.16, 44.93, -73.12, 19.61
 len EQU $-b1
 res DD  len DUP(0)
.code
 _mulps_ex proc
  lea    ESI, a1
  lea    EDI, b1
  movups XMM0, [ESI]
  mulps  XMM0, [EDI]
  movups res, XMM0
  lea    EAX, res
  ret
_mulps_ex endp
 end
```

Алгоритм выполнения процедуры реализован так же, как и для команд параллельного сложения и вычитания, поэтому останавливаться на нем я не буду. Проверить работоспособность процедуры _mulps_ex можно с помощью простой программы на Visual C++ .NET (листинг 13.9).

**Листинг 13.9.** Демонстрационная программа для процедуры из листинга 13.8

```
#include <stdio.h>
extern "C" float* mulps_ex(void);
int main(void)
{
 printf("MULPS example:\n");
 float* pmulps = mulps_ex();
 for (int i1 = 0; i1 < 4;i1++)
  {
   printf("%5.2f ", *pmulps++);
  }
 return 0;
}
```

Напомню, что вызываемая процедура должна быть объявлена с директивой extern, а возвращаемое значение указывает на адрес массива чисел с плавающей точкой (float*) в коротком (32 бита) формате.

Для проверки функционирования команды скалярного умножения можно воспользоваться исходным текстом одной из ранее рассмотренных процедур для скалярного сложения (_addss_ex) или скалярного вычитания (_subss_ex), заменив в них команду addss или subss командой mulss.

Следующая группа команд, работу которых мы проанализируем, — команды параллельного и скалярного деления:

- divps — параллельное деление 128-разрядных операндов; в качестве операнда-источника выступают один из XMM-регистров или 128-разрядная ячейка памяти. Выходным операндом команды должен быть один из XMM-регистров, при этом в качестве делимого выступает операнд-приемник, а в качестве делителя — операнд-источник. Результат помещается в операнд-приемник;
- divss — скалярное деление операндов; младшие двойные слова операндов должны быть числами с плавающей точкой в коротком формате. Результат помещается в операнд-приемник, в качестве которого должен выступать XMM-регистр. Входным операндом могут быть XMM-регистр или 32-разрядная ячейка памяти.

В листинге 13.10 показан программный код, демонстрирующий использование команды divps и реализованный в виде процедуры _divps_ex.

**Листинг 13.10.** Параллельное деление 128-разрядных операндов

```
.686
.model flat
.XMM
option casemap:none
.data
 a1  DD 34.78, -56.07, -129.31, 94.2
```

```
 b1  DD -59.16, 44.93, -73.12, 19.61
 len EQU $-b1
 res DD  len DUP(0)
.code
 _divps_ex proc
  lea    ESI, a1
  lea    EDI, b1
  movups XMM0, [ESI]
  divps  XMM0, [EDI]
  movups res, XMM0
  lea    EAX, res
  ret
_divps_ex endp
 end
```

К группе арифметических команд, помимо тех, что выполняют четыре основных действия, обычно относят команды извлечения квадратного корня, определения максимального/минимального значения и нахождения значения, обратного исходной величине. Перечислим эти команды по порядку:

- sqrtps — параллельное извлечение квадратного корня из упакованных чисел с плавающей точкой. Команда имеет два операнда: источник и приемник. В качестве операнда-источника могут выступать XMM-регистр или 128-разрядная ячейка памяти, в качестве приемника — XMM-регистр;
- sqrtss — скалярное извлечение квадратного корня из упакованного числа с плавающей точкой. В качестве операнда-источника могут выступать 32-разрядная ячейка памяти или XMM-регистр. В том случае, если источником является XMM-регистр, используется младший 32-разрядный операнд, при этом остальные операнды не изменяются. В качестве операнда-приемника должен выступать XMM-регистр;
- maxps — параллельное получение максимального значения для каждой пары упакованных чисел с плавающей точкой. Команда имеет два операнда: источник и приемник. В качестве операнда-источника могут выступать XMM-регистр или 128-разрядная ячейка памяти, в качестве приемника — XMM-регистр, в который помещаются максимальные элементы каждой пары. Содержимое операнда-источника после выполнения операции не изменяется;
- minps — параллельное получение минимального значения для каждой пары упакованных чисел с плавающей точкой. Команда имеет два операнда: источник и приемник. В качестве операнда-источника могут выступать XMM-регистр или 128-разрядная ячейка памяти, в качестве приемника — XMM-регистр, в который помещаются максимальные элементы каждой пары. Содержимое операнда-источника после выполнения операции не изменяется;
- maxss — скалярное получение максимального значения для младшей пары упакованных 32-разрядных операндов приемника и источника. Результат помещается в операнд-приемник, в качестве которого может выступать один из XMM-регистров, при этом его младший операнд замещается максимальным значением. В качестве операнда-источника может выступать 32-разрядное значение в памяти либо младший операнд XMM-регистра. После выполнения операции содержимое операнда-источника остается неизменным;

- minss — скалярное получение минимального значения для младшей пары упакованных 32-разрядных операндов приемника и источника. Результат помещается в операнд-приемник, в качестве которого может выступать один из XMM-регистров, при этом его младший операнд замещается минимальным значением. В качестве операнда-источника может выступать 32-разрядное значение в памяти либо младший операнд XMM-регистра. После выполнения операции содержимое операнда-источника остается неизменным.

Приведу несколько примеров применения этих команд. Например, извлечение квадратного корня из каждого элемента массива чисел с плавающей точкой можно осуществить при помощи демонстрационной процедуры _sqrt_ex, исходный текст которой представлен в листинге 13.11.

**Листинг 13.11.** Извлечение квадратного корня из элементов массива

```
.686
.model flat
.XMM
option casemap:none
.data
  res DD 32 DUP (0)
.code
_sqrt_ex proc
  push   EBP
  mov    EBP, ESP
  mov    EAX, dword ptr [EBP+12]
  shr    EAX, 2
  mov    EBX, 4
  xor    EDX, EDX
  div    EBX
  mov    ECX, EAX
  mov    ESI, dword ptr [EBP+8]
  lea    EDI, res
next:
  sqrtps XMM0, [ESI]
  movups [EDI], XMM0
  add    ESI, 16
  add    EDI, 16
  dec    ECX
  jnz    next
  cmp    EDX, 0
  je     exit
  mov    ECX, EDX
next1:
  sqrtss XMM0, [ESI]
  movss  [EDI], XMM0
  add    ESI, 4
  add    EDI, 4
  dec    ECX
  jnz    next1
exit:
  lea    EAX, res
  pop    EBP
```

```
  ret
_sqrt_ex endp
end
```

Процедура _sqrt_ex в качестве параметров принимает адрес массива чисел с плавающей точкой и размер массива в байтах. Мнемонически объявление процедуры можно представить так:

```
_sqrt_ex(адрес_массива, размер)
```

Процедура возвращает в регистре EAX адрес массива res двойных слов, содержащих значения квадратного корня.

Параметры извлекаются из стека с использованием регистра EBP. Размер исходного массива в общем случае не кратен 16 байт, поэтому часть элементов можно обработать с помощью параллельной команды sqrtps, а часть — с помощью скалярной команды sqrtss. Например, если исходный массив содержит 7 двойных слов (представление коротких чисел с плавающей точкой), то с помощью команды sqrtps можно обработать 4 двойных слова (16 байт), а оставшиеся 3 двойных слова — командой sqrtss.

Следующие команды позволяют выделить количество групп элементов по 4 двойных слова, а также число оставшихся элементов:

```
mov  EAX, dword ptr [EBP+12] ; размер исходного
                             ; массива (в байтах) -> EAX
shr  EAX, 2                  ; преобразовать в количество двойных слов
mov  EBX, 4
xor  EDX, EDX
div  EBX                     ; EAX = количеству 128-разрядных операндов
                             ; EDX = количество оставшихся
                             ; 32-разрядных операндов
```

Далее, команда mov ECX, EAX заполняет в регистр-счетчик цикла для обработки 128-разрядных операндов, а следующие команды загружают в регистры ESI и EDI адреса исходного и результирующего массивов соответственно:

```
mov  ESI, dword ptr [EBP+8]
lea  EDI, res
```

Затем начинается обработка 128-разрядных элементов в цикле next:

```
next:
  sqrtps XMM0, [ESI]
  movups [EDI], XMM0
  . . .
  jnz    next
```

Здесь команда sqrtps формирует результат в регистре XMM0, после чего 4 двойных слова записываются из XMM0 в 4 двойных слова массива res, указатель которого находится в регистре EDI. После этого выполняется переход на следующие адреса в исходном и результирующем массивах и цикл повторяется.

По окончании цикла next проверяется, есть ли еще 32-разрядные элементы:

```
cmp  EDX, 0
je   exit
mov  ECX, EDX
```

Если есть (содержимое EDX отлично от нуля), то переходим к выполнению цикла next1, в котором квадратный корень вычисляется для 32-разрядных элементов при помощи скалярной команды sqrtss:

```
next1:
  sqrtss XMM0. [ESI]
  movss  [EDI]. XMM0
  . . .
  jnz    next1
```

Обратите внимание на то, что пересылка результирующего значения из регистра XMM0 осуществляется командой скалярной пересылки данных:

```
movss [EDI]. XMM0
```

Предпоследняя команда помещает в регистр EAX адрес массива res, в котором находятся вычисленные значения:

```
lea EAX. res
```

После этого происходит выход из процедуры. Замечу, что процедура может обрабатывать массивы очень большого размера (определяется размерностью регистра ECX).

Для тестирования процедуры _sqrt_ex используется программа на Visual C++ .NET (листинг 13.12).

**Листинг 13.12.** Демонстрационная программа для процедуры из листинга 13.11

```
#include <stdio.h>
extern "C" float* sqrt_ex(float* a1. int asize):
int main(void)
{
_declspec(align(16)) float a1[7]= { 44.21. 18.74. 234.65. 82.51. 5249.09. 1.55. 138.02}:
int asize = sizeof(a1):
float* pb1 = sqrt_ex(a1. asize):
printf("SQRTPS-SQRTSS example:\n"):
for (int i1 = 0:i1 < asize / 4: i1++)
 {
  printf("%5.2f ". *pb1++):
 }
 return 0:
}
```

Здесь я хочу отметить один важный момент: некоторые команды SSE-расширения требуют выравнивания данных по 16-байтовой границе (это повышает быстродействие), в противном случае генерируется исключение общей защиты (General Protection Fault, GPF). Такое выравнивание в C++ можно выполнить, указав в объявлении массива ключевые слова

```
_declspec(align 16))
```

Можно обойтись и без выравнивания данных в вызывающей программе, особенно если непонятно, как это сделать. В этом случае вызывающая программа может работать с обычными данными, но при этом следует изменить программный код цикла next самой процедуры _sqrt_ex (показана жирным шрифтом):

```
next:
  movups XMM0, [ESI]
  sqrtps XMM0, XMM0
  movups [EDI], XMM0
  add    ESI, 16
  add    EDI, 16
  dec    ECX
  jnz    next
```

Здесь невыровненные входные данные, находящиеся по адресу в регистре ESI, пересылаются в регистр XMM0 при помощи команды

```
movups XMM0, [ESI]
```

Затем в следующей команде, sqrtps, в качестве входного и выходного операндов просто указываем один и тот же регистр (XMM0). В этом случае исключения общей защиты не возникает.

Вызывающая программа в процессе выполнения выводит на экран такие результаты:

```
SQRTPS-SQRTSS example:
6.65 4.33 15.32 9.08 72.45 1.24 11.75
```

Для демонстрации работы команды поиска максимальных значений из пар элементов разработаем процедуру _max_ex, в которой будем использовать как команду для параллельной обработки (maxps), так и скалярную команду (maxss). В качестве параметров процедура принимает адреса двух массивов и их размер (предполагается, что размеры массивов одинаковы), а возвращает адрес массива, содержащего максимальные элементы. Мнемонически процедуру можно представить так:

```
_max_ex(адрес_массива1, адрес_массива2, размер)
```

Исходный текст процедуры приведен в листинге 13.13.

**Листинг 13.13.** Поиск максимальных элементов в двух массивах

```
.686
.model flat
.XMM
option casemap:none
.data
  res DD 32 DUP (0)
.code
_max_ex proc
  push  EBP
  mov   EBP, ESP
  mov   EAX, dword ptr [EBP+16]
  shr   EAX, 2
  mov   EBX, 4
  xor   EDX, EDX
  div   EBX
  mov   ECX, EAX
  mov   ESI, dword ptr [EBP+8]
  mov   EDI, dword ptr [EBP+12]
  lea   EBX, res
```

продолжение

**Листинг 13.13** (продолжение)

```
next:
  movups XMM0, [ESI]
  movups XMM1, [EDI]
  maxps  XMM0, XMM1
  movups [EBX], XMM0
  add    ESI, 16
  add    EDI, 16
  add    EBX, 16
  dec    ECX
  jnz    next
  cmp    EDX, 0
  je     exit
  mov    ECX, EDX
next1:
  movss  XMM0, [ESI]
  movss  XMM1, [EDI]
  maxss  XMM0, XMM1
  movss  [EBX], XMM0
  add    ESI, 4
  add    EDI, 4
  add    EBX, 4
  dec    ECX
  jnz    next1
exit:
  lea    EAX, res
  pop    EBP
  ret
_max_ex endp
end
```

Большая часть программного кода процедуры нам знакома из предыдущих примеров, поэтому остановлюсь на ключевых аспектах.

Процедура принимает три параметра с использованием регистра EBP: адрес первого массива вещественных чисел ([EBP+8]), адрес второго массива ([EBP+12]) и размер массивов в байтах ([EBP+16]). Кроме того, предполагается, что размер обоих массивов одинаков. Адреса этих массивов вместе с адресом массива res, куда будет помещен результат, загружаются в регистры ESI, EDI и EBX соответственно:

```
mov  ESI, dword ptr [EBP+8]
mov  EDI, dword ptr [EBP+12]
lea  EBX, res
```

Собственно вычисления выполняются, как и в предыдущих примерах, в двух циклах: next и next1. В цикле next происходят параллельные операции по определению максимального значения в 32-разрядных парах 128-разрядных элементов:

```
next:
  movups XMM0, [ESI]
  movups XMM1, [EDI]
  maxps  XMM0, XMM1
  movups [EBX], XMM0
  . . .
  jnz    next
```

Здесь 128-разрядные операнды загружаются в регистры XMM0 и XMM1, после чего вычисляются максимальные значения командой

```
maxps XMM0, XMM1
```

В цикле next1 обрабатываются одиночные 32-разрядные элементы массивов с использованием скалярной команды maxss:

```
next1:
 movss  XMM0, [ESI]
 movss  XMM1, [EDI]
 maxss  XMM0, XMM1
 movss  [EBX], XMM0
 . . .
 jnz    next1
```

По окончании вычислений адрес массива res, где находятся результаты вычислений, помещается в регистр EAX, после чего происходит выход из процедуры.

Используя программный код процедуры _max_ex, легко создать процедуру для вычисления минимальных значений в парах элементов массивов. Для этого достаточно заменить команды вычисления максимума соответствующими командами вычисления минимума (maxps заменить на minps, а maxss — на minss).

Последняя группа команд, которые относятся к арифметическим и которые мы сейчас рассмотрим, — это команды вычисления обратных значений. Как и большинство SSE-команд, они оперируют либо четырьмя упакованными 32-разрядными значениями (параллельные), либо младшими 32-разрядными операндами (скалярные). Команды вычисления обратных значений обеспечивают точность не менее 11 бит. Это означает, что максимальное значение относительной погрешности будет менее чем $1{,}5 \times 2^{-12}$. Для лучшего понимания можно выразить значение относительной погрешности формулой

$$\frac{|\text{точное значение} - \text{приблизительное значение}|}{\text{приблизительное значение}} \le 1{,}5 \times 2^{-12}$$

К командам этой подгруппы относятся:

- rcpps — параллельное вычисление обратных значений упакованных операндов. Если $X$ — значение одного из 32-разрядных операндов, то вычисляется $1/X$. В качестве входного операнда могут выступать XMM-регистр или 128-разрядная ячейка памяти, а выходным операндом должен быть XMM-регистр;
- rcpss — скалярное вычисление обратного значения младшего 32-разрядного операнда. В качестве входного операнда могут выступать XMM-регистр или 32-разрядная ячейка памяти, а выходным операндом должен быть XMM-регистр. Операция не затрагивает содержимого старших операндов XMM-регистра;
- rsqrtps — параллельное вычисление обратных значений квадратного корня упакованных операндов. Если $X$ — значение одного из 32-разрядных операндов, то вычисляется $1/\sqrt{X}$. В качестве входного операнда могут выступать XMM-регистр или 128-разрядная ячейка памяти, а выходным операндом должен быть XMM-регистр;

- rsqrtss — скалярное вычисление обратного значения квадратного корня младшего 32-разрядного операнда. В качестве входного операнда могут выступать XMM-регистр или 32-разрядная ячейка памяти, а выходным операндом должен быть XMM-регистр. Операция не затрагивает содержимого старших операндов XMM-регистра.

Рассмотрим несколько примеров работы команд, вычисляющих обратные значения. Первый пример демонстрирует применение команд rcpps и rcpss и реализован в виде процедуры _rcp_ex. Процедура в качестве параметров принимает адрес массива чисел с плавающей точкой и размер массива (параметры извлекаются при помощи регистра EBP).

Хочу обратить внимание на то, что данные массива желательно выровнять по 16-байтовой границе. Процедура возвращает в регистре EAX адрес массива, в который помещаются вычисленные обратные значения. Исходный текст процедуры _rcp_ex показан в листинге 13.14.

**Листинг 13.14.** Вычисление обратных значений чисел

```
.686
.model flat
.XMM
option casemap:none
.data
  res DD 32 DUP (0)
.code
_rcp_ex proc
  push   EBP
  mov    EBP, ESP
  mov    EAX, dword ptr [EBP+12]
  shr    EAX, 2
  mov    EBX, 4
  xor    EDX, EDX
  div    EBX
  mov    ECX, EAX
  mov    ESI, dword ptr [EBP+8]
  lea    EDI, res
next:
  rcpps XMM0, [ESI]
  movups [EDI], XMM0
  add    ESI, 16
  add    EDI, 16
  dec    ECX
  jnz    next
  cmp    EDX, 0
  je     exit
  mov    ECX, EDX
next1:
  rcpss XMM0, [ESI]
  movss  [EDI], XMM0
  add    ESI, 4
  add    EDI, 4
  dec    ECX
  jnz    next1
```

```
exit:
  lea    EAX, res
  pop    EBP
  ret
_rcp_ex endp
end
```

Здесь, как и в предыдущих примерах, отдельно обрабатываются 128-разрядные элементы и младшие двойные слова. В цикле next выполняется параллельное вычисление обратных значений при помощи команды rcpps:

```
next:
  rcpps XMM0, [ESI]
  movups [EDI], XMM0
  . . .
  jnz    next
```

Отдельные 32-разрядные операнды исходного массива обрабатываются в цикле next1:

```
next1:
  rcpss XMM0, [ESI]
  movss  [EDI], XMM0
  . . .
  jnz    next1
```

Для вычисления обратных значений квадратного корня программный код процедуры _rcp_ex можно модифицировать, заменив команды rcpps и rcpss командами rsqrtps и rsqrtss.

# 13.3. Команды сравнения

Команды сравнения позволяют определять соответствие операндов указанным условиям и, в зависимости от результата сравнения, устанавливают в нужном элементе операнда-приемника двоичные нули (если условие не выполняется) или двоичные единицы (если условие выполняется). Команды сравнения могут выполняться параллельно над упакованными операндами или скалярно над младшими двойными словами. Все команды имеют два операнда: в качестве входного операнда, или операнда-источника, могут выступать XMM-регистр или ячейка памяти (128-разрядная для параллельных команд и 32-разрядная для скалярных). Выходным операндом, или операндом-приемником, может быть только один из XMM-регистров, в котором задействованы либо 128 бит (параллельные команды), либо младшие 32 бита (скалярные команды).

Группа параллельных команд:

- cmpeqps — проверяет условие равенства 32-разрядных упакованных операндов и устанавливает соответствующие значения в операнде-приемнике. Содержимое операнда-источника после выполнения операции не изменяется;
- cmpltps — проверяет условие «меньше» (less-than) для 32-разрядных упакованных операндов и устанавливает соответствующие значения в операнде-приемнике. Содержимое операнда-источника после выполнения операции не изменяется;

- cmpleps — проверяет условие «меньше или равно» (less-than-or-equal) для 32-разрядных упакованных операндов и устанавливает соответствующие значения в операнде-приемнике. Содержимое операнда-источника после выполнения операции не изменяется;
- cmpunordps — проверяет условие «неупорядоченности» (unordered) для 32-разрядных упакованных операндов и устанавливает соответствующие значения в операнде-приемнике. Содержимое операнда-источника после выполнения операции не изменяется;
- cmpneqps — проверяет условие «не равно» (not-equal-to) для 32-разрядных упакованных операндов и устанавливает соответствующие значения в операнде-приемнике. Содержимое операнда-источника после выполнения операции не изменяется;
- cmpnltps — проверяет условие «не меньше» (not-less-than) для 32-разрядных упакованных операндов и устанавливает соответствующие значения в операнде-приемнике. Содержимое операнда-источника после выполнения операции не изменяется;
- cmpnleps — проверяет условие «не меньше или равно» (not-less-than-or-equal-to) для 32-разрядных упакованных операндов и устанавливает соответствующие значения в операнде-приемнике. Содержимое операнда-источника после выполнения операции не изменяется;
- cmpordps — проверяет условие «упорядоченности» (ordered) для 32-разрядных упакованных операндов и устанавливает соответствующие значения в операнде-приемнике. Содержимое операнда-источника после выполнения операции не изменяется.

Группа скалярных команд:

- cmpeqss — проверяет условие равенства младших 32-разрядных операндов и устанавливает соответствующее значение в младшем двойном слове операнда-приемника. Содержимое операнда-источника после выполнения операции не изменяется;
- cmpltps — проверяет условие «меньше» (less-than) для младших 32-разрядных упакованных операндов и устанавливает соответствующее значение в младшем двойном слове операнда-приемника. Содержимое операнда-источника после выполнения операции не изменяется;
- cmpleps — проверяет условие «меньше или равно» (less-than-or-equal) для младших 32-разрядных упакованных операндов и устанавливает соответствующее значение в младшем двойном слове операнда-приемника. Содержимое операнда-источника после выполнения операции не изменяется;
- cmpunordps — проверяет условие «неупорядоченности» (unordered) для младших 32-разрядных упакованных операндов и устанавливает соответствующее значение в младшем двойном слове операнда-приемника. Содержимое операнда-источника после выполнения операции не изменяется;

- cmpneqps — проверяет условие «не равно» (not-equal-to) для младших 32-разрядных упакованных операндов и устанавливает соответствующее значение в младшем двойном слове операнда-приемника. Содержимое операнда-источника после выполнения операции не изменяется;
- cmpnltps — проверяет условие «не меньше» (not-less-than) для младших 32-разрядных упакованных операндов и устанавливает соответствующее значение в младшем двойном слове операнда-приемника. Содержимое операнда-источника после выполнения операции не изменяется;
- cmpnleps — проверяет условие «не меньше или равно» (not-less-than-or-equal-to) для младших 32-разрядных упакованных операндов и устанавливает соответствующее значение в младшем двойном слове операнда-приемника. Содержимое операнда-источника после выполнения операции не изменяется;
- cmpordps — проверяет условие «упорядоченности» (ordered) для младших 32-разрядных упакованных операндов и устанавливает соответствующее значение в младшем двойном слове операнда-приемника. Содержимое операнда-источника после выполнения операции не изменяется.

В листинге 13.15 приводится несложный пример использования двух команд (cmpeqps и cmpeqss) в процедуре _cmp_ex, выполняющей сравнение элементов двух массивов, элементами которых являются числа с плавающей точкой в коротком формате. Процедура принимает три входных параметра: адреса двух массивов и размер любого из них в байтах (предполагается, что размер массивов одинаков). Параметры извлекаются посредством регистра EBP, а результат процедура возвращает в регистре EAX (адрес целочисленного массива, содержащего результаты сравнения).

**Листинг 13.15.** Сравнение элементов массивов

```
.686
.model flat
.XMM
option casemap:none
.data
  res DD 32 DUP (0)
.code
_cmp_ex proc
  push    EBP
  mov     EBP, ESP
  mov     EAX, dword ptr [EBP+16]
  shr     EAX, 2
  mov     EBX, 4
  xor     EDX, EDX
  div     EBX
  mov     ECX, EAX
  mov     ESI, dword ptr [EBP+8]
  mov     EDI, dword ptr [EBP+12]
  lea     EBX, res
```

продолжение ⇗

**Листинг 13.15** (продолжение)

```
next:
  movups  XMM0, [ESI]
  cmpeqps XMM0, [EDI]
  movups  [EBX], XMM0
  add     ESI, 16
  add     EDI, 16
  add     EBX, 16
  dec     ECX
  jnz     next
  cmp     EDX, 0
  je      exit
  mov     ECX, EDX
next1:
  movss   XMM0, [ESI]
  cmpeqss XMM0, [EDI]
  movss   [EBX], XMM0
  add     ESI, 4
  add     EDI, 4
  add     EBX, 4
  dec     ECX
  jnz     next1
exit:
  lea     EAX, res
  pop     EBP
  ret
_cmp_ex endp
end
```

В этой процедуре обработка элементов массивов, как и в предыдущих примерах, разбивается на два этапа. В начале сравниваются 128-разрядные элементы при помощи команды cmpeqps в цикле next:

```
next:
  movups  XMM0, [ESI]
  cmpeqps XMM0, [EDI]
  movups  [EBX], XMM0
  . . .
  jnz     next
```

Затем в цикле next1 сравниваются оставшиеся отдельные 32-разрядные элементы:

```
next1:
  movss   XMM0, [ESI]
  cmpeqss XMM0, [EDI]
  movss   [EBX], XMM0
  . . .
  jnz     next1
```

После обработки всех элементов массив res будет содержать двойные слова, каждое из которых состоит либо из одних нулей, либо из одних единиц. Процедура возвращает адрес этого массива в регистре EAX.

Для тестирования процедуры _cmp_ex была создана простая программа на Visual C++.NET, выводящая результат сравнения на экран (листинг 13.16).

**Листинг 13.16.** Демонстрационная программа для процедуры из листинга 13.15

```
#include <stdio.h>
extern "C" unsigned int* cmp_ex(float* a1, float* a2, int asize);
int main(void)
{
_declspec(align(16)) float a1[7] = { 44.22, 18.74, 0.66, 82.50, 524.09, 11.55, 38.02};
_declspec(align(16)) float a2[7] = { 44.22, 18.74, 0.65, 81.51, 524.09, 11.55, 38.02};
int asize = sizeof(a1);
unsigned int* pcmp = cmp_ex(a1, a2, asize);
printf("CMPEQPS-CMPEQSS example:\n");
for (int i1 = 0;i1 < asize / 4; i1++)
 {
 printf("%Xh ", *pcmp++);
 }
return 0;
}
```

Как обычно, процедура _cmp_ex должна быть объявлена с директивой extern. Обратите внимание на тип возвращаемого процедурой значения — это указатель на беззнаковое целое число (unsigned int*). Что же касается ключевых слов __declspec(align(16)), то их назначение мы уже рассматривали. При указанных значениях элементов массивов результат, выводимый на экран, будет выглядеть так:

```
CMPEQPS-CMPEQSS example:
FFFFFFFFh FFFFFFFFh 0h 0h FFFFFFFFh FFFFFFFFh FFFFFFFFh
```

Это означает, что элементы с индексами 2 и 3 массивов a1 и a2 не равны между собой.

В группу команд сравнения входят еще две команды: comiss и ucomiss. Команды имеют два операнда и выполняют скалярное сравнение младших 32-разрядных операндов. Особенностью этих команд является то, что содержимое обоих операндов после выполнения операции сравнения остается неизменным, но в регистре флагов EFLAGS процессора определенным образом устанавливаются флаги ZF, PF и CF, а флаги OF, SF и AF сбрасываются в 0. В качестве входных операндов обеих команд могут выступать XMM-регистры или 32-разрядные переменные в памяти, выходными операндами могут быть только XMM-регистры.

Команды ucomiss и comiss отличаются тем, что генерируют исключительные ситуации для различных форматов не-чисел (NAN). Эти команды очень удобны при организации ветвлений в программах, поскольку по состоянию флагов позволяют интерпретировать результат сравнения. В табл. 13.1 приводится соотношение между сравниваемыми операндами и устанавливаемыми флагами.

**Таблица 13.1.** Результат сравнения и состояние флагов

| Результат сравнения операндов op1 и op2 | Флаг ZF | Флаг PF | Флаг CF |
|---|---|---|---|
| Операнды неупорядочены (unordered) | 1 | 1 | 1 |
| op1 < op2 | 0 | 0 | 1 |
| op1 > op2 | 0 | 0 | 0 |
| op1 = op2 | 1 | 0 | 0 |

В листинге 13.17 приводится пример использования команды comiss в процедуре, выполняющей сравнение двух чисел с плавающей точкой (назовем их a1 и a2), адреса которых являются для нее входными параметрами. Процедура называется _comiss_ex и по завершении возвращает в регистре EAX адрес строки, содержащей сообщение о результате сравнения.

**Листинг 13.17.** Сравнение чисел с плавающей точкой, находящихся в двух массивах

```
.686
.model flat
option casemap: none
.XMM
.data
equal      DB "a1 = a2", 0
not_equal  DB "a1 not equal a2", 0
.code
_comiss_ex proc
  push   EBP
  mov    EBP, ESP
  mov    ESI, dword ptr [EBP+8]
  mov    EDI, dword ptr [EBP+12]
  movss  XMM0, [ESI]
  comiss XMM0, [EDI]
  lahf
  and    AH, 45h
  cmp    AH, 40h
  je     ops_equal
  lea    EAX, not_equal
  jmp    exit
ops_equal:
  lea    EAX, equal
exit:
  pop    EBP
  ret
_comiss_ex endp
end
```

Исходный текст процедуры несложен, поэтому подробно останавливаться на нем я не буду. Выделю лишь три команды, которые, собственно, и выполняют основную работу:

```
movss  XMM0, [ESI]
comiss XMM0, [EDI]
lahf
and    AH, 45h
cmp    AH, 40h
je     ops_equal
```

Здесь команда comiss выполняет сравнение младшего 32-разрядного числа из регистра XMM0, представляющего собой значение первого операнда, со вторым операндом, находящимся по адресу в регистре EDI, при этом младшее 32-разрядное число оказывается в регистре XMM0 с помощью команды

```
movss XMM0, [ESI]
```

Затем содержимое регистра EFLAGS помещается в регистр AH командой lahf. Для определения соотношения между операндами требуется анализ 3 бит регистра AH, соответствующих флагам ZF, PF и CF (рис. 13.14).

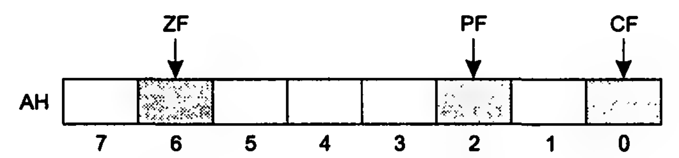



**Рис. 13.14.** Содержимое регистра AH после выполнения команды lahf

Предварительно замаскируем ненужные биты регистра AH при помощи команды

```
and AH, 45h
```

Далее, сравним содержимое регистра со значением 40h (признак равенства операндов, как показано в табл. 13.1), после чего выполним переход на нужную ветвь процедуры командой

```
jz ops_equal
```

Если операнды равны, процедура возвращает строку equal, если не равны — строку not_equal. Тестирование процедуры _comiss_ex можно выполнить с помощью следующей программы из листинга 13.18.

**Листинг 13.18.** Демонстрационная программа для процедуры из листинга 13.17

```
#include <stdio.h>
extern "C" char* comiss_ex(float* a1, float* a2);
int main(void)
{
 float a1 = -34.71;
 float a2 = -34.71;
 printf("COMISS example: %s\n", comiss_ex(&a1, &a2));
 return 0;
}
```

При желании случаи «больше» и «меньше» для только что рассмотренной задачи читатель сможет проанализировать самостоятельно.

Перейдем к следующей группе команд, позволяющих выполнять взаимное преобразование значений, представленных тремя типами данных: целыми числами в формате MMX, числами с плавающей точкой в коротком формате (SSE) и обычными 32-разрядными числами.

# 13.4. Команды преобразования

Команды преобразования могут выполняться, как и большинство остальных SSE-команд, в параллельном и скалярном контекстах. К этой группе команд относятся:

- cvtps2pi — параллельное преобразование двух младших упакованных 32-разрядных чисел с плавающей точкой в коротком формате в два 32-разрядных целых числа. Команда имеет два операнда. В качестве входного операнда

могут выступать XMM-регистр или 128-разрядная ячейка памяти. Выходным операндом может служить один из регистров MMX. На результат преобразования влияет установка битов поля **rc**, определяющего режим округления, регистра управления/состояния (MXCSR);

- cvtss2si — скалярное преобразование младшего 32-разрядного числа с плавающей точкой в коротком формате в 32-разрядное целое число. Команда имеет два операнда. В качестве входного операнда могут выступать XMM-регистр или 128-разрядная ячейка памяти. Выходным операндом может служить один из 32-разрядных регистров общего назначения. Как и для команды cvtps2pi, на результат преобразования влияет установка битов поля **rc**, определяющего режим округления, регистра управления/состояния (MXCSR);
- cvttps2pi — параллельное преобразование двух младших упакованных 32-разрядных чисел с плавающей точкой в коротком формате в два 32-разрядных целых числа путем отсечения дробной части исходных операндов. Команда имеет два операнда. В качестве входного операнда могут выступать XMM-регистр или 128-разрядная ячейка памяти. Выходным операндом может служить один из MMX-регистров. Установка битов поля **rc**, определяющего режим округления, регистра управления/состояния (MXCSR) на результат не влияет;
- cvttss2si — скалярное преобразование младшего 32-разрядного числа с плавающей точкой в коротком формате в 32-разрядное целое число путем отсечения дробной части входного операнда. Команда имеет два операнда. В качестве входного операнда могут выступать XMM-регистр или 128-разрядная ячейка памяти, а выходным операндом может служить один из 32-разрядных регистров общего назначения. Установка битов поля **rc**, определяющего режим округления, регистра управления/состояния (MXCSR) на результат не влияет;
- cvtpi2ps — преобразование двух целых 32-разрядных чисел со знаком в два 32-разрядных числа с плавающей точкой в коротком формате. Команда имеет два операнда. В качестве входного операнда, или операнда-источника, может выступать MMX-регистр или 64-разрядный операнд в памяти. Выходным операндом может служить только XMM-регистр. Результат преобразования помещается в младшие два 32-разрядных элемента XMM-регистра, старшие два элемента остаются без изменений. На результат преобразования влияет установка битов поля **rc**, определяющего режим округления, регистра управления/состояния (MXCSR);
- cvtsi2ss — преобразование 32-разрядного числа в целочисленном формате в 32-разрядное число с плавающей точкой в коротком формате. Команда имеет два операнда. В качестве входного операнда, или операнда-источника, может выступать 32-разрядный регистр общего назначения или операнд в памяти. Выходным операндом может служить только XMM-регистр. Результат преобразования помещается в младший 32-разрядный элемент XMM-регистра, при этом три старших элемента остаются без изменений. На результат преобразования влияет установка битов поля **rc**, определяющего режим округления, регистра управления/состояния (MXCSR).

После описания команд преобразования рассмотрим пример их практического применения — продемонстрируем работу команд cvtps2pi и cvtss2si. В листинге 13.19 показан программный код процедуры _cvtps2pi_ex, в которой показан процесс преобразования элементов массива чисел с плавающей точкой в целочисленные значения. Процедура принимает два параметра: адрес исходного массива вещественных чисел и его размер в байтах. Все параметры извлекаются с помощью регистра EBP. При этом адрес исходного массива вещественных чисел передается в [EBP+8], а размер массива в байтах — в [EBP+12]. Результат преобразования возвращается в регистре EAX в виде адреса массива целых чисел.

**Листинг 13.19.** Преобразование элементов массива чисел с плавающей точкой в целочисленные значения

```
.686
.model flat
.XMM
option casemap:none
.data
  res DD 32 DUP (0)
.code
_cvtps2pi_ex proc
  push     EBP
  mov      EBP, ESP
  mov      EAX, dword ptr [EBP+12]
  shr      EAX, 2
  mov      EBX, 2
  xor      EDX, EDX
  div      EBX
  mov      ECX, EAX
  mov      ESI, dword ptr [EBP+8]
  lea      EDI, res
next:
  movlps   XMM0, [ESI]
  cvtps2pi MM0, XMM0
  movq     [EDI], MM0
  add      ESI, 8
  add      EDI, 8
  dec      ECX
  jnz      next
  cmp      EDX, 0
  je       exit
  mov      ECX, EDX
next1:
  movss    XMM0, [ESI]
  cvtss2si EAX, XMM0
  mov      [EDI], EAX
  add      ESI, 4
  add      EDI, 4
  dec      ECX
  jnz      next1
exit:
  lea      EAX, res
  pop      EBP
  ret
_cvtps2pi_ex endp
end
```

Работа этой процедуры построена по тому же принципу, что и в рассмотренных ранее примерах: элементы массива обрабатываются группами с помощью параллельных команд, а те элементы, которые остались вне этих групп, обрабатываются скалярными командами. В данном случае мы оперируем 64-разрядными операндами (два двойных слова), которые обрабатываются командой cvtps2pi с 32-разрядными операндами (команда cvtss2si).

Таким образом, в программе работают два цикла: в одном обрабатываются 64-разрядные элементы, в другом — 32-разрядные. Рассмотрим практическую реализацию кода. Вначале определяем значения счетчиков циклов:

```
mov  EAX, dword ptr [EBP+12]
shr  EAX, 2
mov  EBX, 2
xor  EDX, EDX
div  EBX
```

После выполнения этого фрагмента кода регистр EAX будет содержать количество 64-разрядных элементов, а регистр EDX — количество оставшихся 32-разрядных операндов. Следующая команда помещает значение счетчика цикла в регистр ECX:

```
mov ECX, EAX
```

Затем начинается обработка 64-разрядных элементов в цикле next:

```
next:
  movlps   XMM0, [ESI]
  cvtps2pi MM0, XMM0
  movq     [EDI], MM0
  . . .
  jnz      next
```

Здесь элементы исходного массива загружаются в младшие два двойных слова регистра XMM0 командой

```
movlps XMM0, [ESI]
```

Затем следующая команда преобразует два 32-разрядных числа с плавающей точкой, находящиеся в младших разрядах XMM0, в целочисленный формат и помещает два 32-разрядных целых числа в регистр MM0:

```
cvtps2pi MM0, XMM0
```

Содержимое регистра MM0 помещается в массив res командой

```
movq [EDI], MM0
```

В массиве res и хранится результат преобразования.

После завершения цикла next, если остались необработанные одиночные 32-разрядные элементы (содержимое регистра EDX отлично от нуля), начинается выполнение цикла next1:

```
next1:
  movss    XMM0, [ESI]
  cvtss2si EAX, XMM0
  mov      [EDI], EAX
  . . .
  jnz      next1
```

Обработка оставшихся элементов исходного массива осуществляется при помощи скалярных команд, оперирующих с 32-разрядными элементами. Здесь команда movss XMM0, [ESI] загружает 32-разрядное число с плавающей точкой в младшее двойное слово регистра XMM0, после чего команда cvtss2si EAX, XMM0 преобразует число к целочисленному формату и сохраняет его в регистр EAX. Замечу, что в качестве регистра-приемника в этой команде можно использовать и другой регистр, например EBX.

После окончания преобразования всех элементов исходного массива в целочисленные значения команда lea EAX, res помещает адрес массива, где хранятся результаты, в регистр EAX, после чего происходит выход из процедуры.

Работоспособность процедуры _cvtps2pi_ex можно легко проверить с помощью программы на Visual C++ .NET (листинг 13.20).

**Листинг 13.20.** Демонстрационная программа для процедуры из листинга 13.19

```
#include <stdio.h>
extern "C" int* cvtps2pi_ex(float* a1, int asize);
int main(void)
{
__declspec(align(16)) float a1[9] = {-56.3, -11.49, 23.04, 477.59, -6.51, 45.81, -781.21,
233.76};
 int asize = sizeof(a1);
 printf("CVTPS2PI example:\n");
 int* pi1 = cvtps2pi_ex(a1, asize);
 for (int i1 = 0; i1 < 8;i1++)
 {
  printf("%d ", *pi1++);
 }
return 0;
}
```

При указанных значениях элементов массива a1 программа выводит такой результат:

```
CVTPS2PI example:
-56 -11 23 478 -7 46 -781 234
```

Как видно из результата, вещественные числа массива a1 были округлены к ближайшему целому числу. Обычно этот режим задается по умолчанию при инициализации процессора (поле **rc** регистра MXCSR). Во многих случаях требуется изменить режим округления, например выполнить округление к большему значению. Сейчас мы посмотрим, как можно установить режим округления программными средствами.

В начале главы мы рассматривали назначение полей регистра управления/состояния (MXCSR). Напомню, что биты 13–14 (поле **rc**) определяют режим округления. По умолчанию это поле содержит значение 00, то есть округление выполняется в сторону ближайшего числа. Наш следующий пример продемонстрирует обработку данных в случае, когда принят режим округления в меньшую сторону (биты 13–14 регистра MXCSR должны быть установлены в 1 и 0 соответственно).

Воспользуемся программным кодом только что рассмотренной процедуры _cvtps2pi_ex и модифицируем его. В новой процедуре (она называется _cvt_mod)

перед обработкой массива вещественных чисел устанавливается режим округления к меньшему числу. После этого выполняются те же операции, что и в процедуре _cvtps2pi_ex. Исходный текст процедуры _cvt_mod приведен в листинге 13.21.

**Листинг 13.21.** Округление элементов массива чисел с плавающей точкой к меньшему целому числу

```
.686
.model flat
.XMM
option casemap:none
.data
  res        DD 32 DUP (0)
  state      label dword
  state_low  DW 0
             DW 0
.code
_cvt_mod proc
  push     EBP
  mov      EBP, ESP
  mov      EAX, dword ptr [EBP+12]
  shr      EAX, 2
  mov      EBX, 2
  xor      EDX, EDX
  div      EBX
  mov      ECX, EAX
  mov      ESI, dword ptr [EBP+8]
  lea      EDI, res
  stmxcsr  state
  or       state_low, 2000h
  ldmxcsr  state
next:
  movlps   XMM0, [ESI]
  cvtps2pi MM0, XMM0
  movq     [EDI], MM0
  add      ESI, 8
  add      EDI, 8
  dec      ECX
  jnz      next
  cmp      EDX, 0
  je       exit
  mov      ECX, EDX
next1:
  movss    XMM0, [ESI]
  cvtss2si EAX, XMM0
  mov      [EDI], EAX
  add      ESI, 4
  add      EDI, 4
  dec      ECX
  jnz      next1
exit:
  lea      EAX, res
  pop      EBP
  ret
_cvt_mod endp
end
```

Исходный текст процедуры _cvt_mod во многом похож на тот, что мы рассматривали при анализе предыдущего примера, поэтому я остановлюсь только на сделанных изменениях.

Начнем с области данных. Здесь добавлены следующие поля:

```
state      label dword
state_low  DW 0
           DW 0
```

Метка state указывает на область данных размером в одно двойное слово, в которой будет храниться содержимое регистра MXCSR. Поскольку нас интересуют только младшие 16 бит состояния, то здесь же определена переменная state_low, с которой и будут проводиться необходимые манипуляции. Такая форма записи выбрана для того, чтобы сделать более понятной последовательность команд для задания нового режима округления.

В секции кода добавлены команды, задающие режим округления в сторону меньшего значения:

```
stmxcsr  state
or       state_low, 2000h
ldmxcsr  state
```

Первая из этих команд, stmxcsr state, сохраняет содержимое регистра MXCSR в области данных state. Далее, нам нужно присвоить битам 13–14 младшего слова значение 10, не затрагивая остальные разряды. Это делает команда

```
or       state_low, 2000h
```

Наконец, загрузим новое двойное слово состояния обратно в регистр MXCSR с помощью команды ldmxcsr state. После выполнения этой команды округление будет выполняться в меньшую сторону.

Для проверки работоспособности процедуры можно выполнить программу, написанную на C++. Результат должен выглядеть так:

```
CVTPS2PI example:
-57 -12 23 477 -7 45 -782 233
```

Если нужно установить режим округления к большему числу, то можно воспользоваться следующими командами:

```
stmxcsr  state
or       state_low, 4000h
ldmxcsr  state
```

Если нужно просто отбросить дробную часть, то следует выполнить команды

```
stmxcsr  state
or       state_low, 6000h
ldmxcsr  state
```

На этом закончим анализ команд преобразования. При желании читатели могут протестировать возможности преобразования из целочисленного формата в формат чисел с плавающей точкой.

Следующая группа команд SSE-расширения, которую мы рассмотрим, — логические команды.

## 13.5. Логические команды

В отличие от большинства команд SSE-расширения, все логические команды являются параллельными и позволяют выполнять операции логического И, ИЛИ, И-НЕ, исключающего ИЛИ над отдельными парами битов операндов:

- andps — параллельная операция логического И над парами битов упакованных чисел с плавающей точкой операнда-источника и операнда-приемника. Команда имеет два операнда: в качестве входного операнда (источника) могут выступать XMM-регистр или 128-разрядная ячейка памяти, а в качестве выходного (приемника) — XMM-регистр. После выполнения команды содержимое операнда-источника не изменяется, а результат помещается в операнд-приемник;
- andnps — параллельная операция логического И-НЕ над парами битов упакованных чисел с плавающей точкой операнда-источника и операнда-приемника. Команда имеет два операнда: в качестве входного операнда (источника) могут выступать XMM-регистр или 128-разрядная ячейка памяти, а в качестве выходного (приемника) — XMM-регистр. После выполнения команды содержимое операнда-источника не изменяется, а результат помещается в операнд-приемник;
- orps — параллельная операция логического ИЛИ над парами битов упакованных чисел с плавающей точкой операнда-источника и операнда-приемника. Команда имеет два операнда: в качестве входного операнда (источника) могут выступать XMM-регистр или 128-разрядная ячейка памяти, а в качестве выходного (приемника) — XMM-регистр. После выполнения команды содержимое операнда-источника не изменяется, а результат помещается в операнд-приемник;
- xorps — параллельная операция логического исключающего ИЛИ над парами битов упакованных чисел с плавающей точкой операнда-источника и операнда-приемника. Команда имеет два операнда: в качестве входного операнда (источника) могут выступать XMM-регистр или 128-разрядная ячейка памяти, а в качестве выходного (приемника) — XMM-регистр. После выполнения команды содержимое операнда-источника не изменяется, а результат помещается в операнд-приемник.

Логические команды могут использоваться для изменения знака числа, поиска абсолютного значения числа (модуля) и организации ветвлений в программах. Далее показан пример программного кода, выполняющего поиск абсолютного значения чисел в массиве и реализованного в виде процедуры _mod_ex. Помимо других команд в процедуре используется команда xorps.

Процедура принимает один параметр — адрес массива чисел с плавающей точкой. Для упрощения полагаем, что массив состоит из 4 двойных слов. Результат вычисления возвращается в регистре EAX, в который помещается адрес массива с абсолютными значениями. Исходный текст процедуры _mod_ex показан в листинге 13.22.

**Листинг 13.22.** Поиск абсолютных значений элементов массива чисел с плавающей точкой

```
.686
.model flat
.XMM
option casemap: none
.data
  res DD 4 DUP(0)
.code
_mod_ex proc
  push   EBP
  mov    EBP, ESP
  mov    ESI, dword ptr [EBP+8]
  lea    EDI, res
  movups XMM0, [ESI]
  xorps  XMM1, XMM1
  subps  XMM1, XMM0
  maxps  XMM1, XMM0
  movups [EDI], XMM1
  lea    EAX, res
  pop    EBP
  ret
_mod_ex endp
end
```

Алгоритм работы процедуры основан на поиске максимальных элементов в парах 32-разрядных упакованных чисел, находящихся в регистрах XMM0 и XMM1. При этом один из регистров будет содержать исходные значения, а другой — значения с противоположным знаком. Используя команду maxps, можно выбрать максимальное из двух значение, которое в любом случае является положительным числом.

Программный код процедуры несложен. Вначале исходное значение 128-разрядного операнда помещается в регистр XMM0 командой

```
movups XMM0, [ESI]
```

Далее, следующая команда обнуляет значения упакованных операндов регистра XMM1:

```
xorps XMM1, XMM1
```

После выполнения команды subps XMM1, XMM0 регистр XMM1 будет содержать значения, противоположные по знаку исходным. Наконец, команда maxps XMM1, XMM0 вычисляет большие значения из пар 32-разрядных операндов и помещает их в регистр XMM1.

Результат вычисления сохраняется в массиве res при помощи команды

```
movups [EDI], XMM1
```

Предпоследняя команда сохраняет адрес массива res с результатами:

```
lea EAX, res
```

После этого происходит выход из процедуры.

## 13.6. Команды управления состоянием

К группе команд управления состоянием относятся команды, выполняющие загрузку/сохранение регистров состояния и управления:

- ldmxcsr — загрузка регистра управления/состояния (MXCSR) содержимым 32-разрядной ячейки памяти, которая и является единственным операндом;
- stmxcsr — сохранение содержимого регистра управления/состояния (MXCSR) в 32-разрядной ячейке памяти, которая и является единственным операндом;
- fxrstor — загрузка предварительно сохраненного состояния сопроцессора, MMX- и SEE-расширения из области памяти размером 512 байт. В качестве операнда выступает адрес области памяти, который должен быть выровнен по 16-байтовой границе;
- fxsave — сохранение состояния сопроцессора, MMX- и SSE-расширения в область памяти размером в 512 байт. В качестве операнда выступает адрес области памяти.

## 13.7. Команды распаковки данных

В группу команд распаковки данных входят две команды:

- unpckhps — параллельное перемещение старших двойных слов из операнда-источника и операнда-приемника в операнд-приемник. При этом два старших двойных слова операнда-источника становятся старшими двойными словами в 64-разрядных элементах операнда-приемника, а два старших двойных слова операнда-приемника — младшими двойными словами в 64-разрядных элементах операнда-приемника. Входным операндом (источником) могут выступать XMM-регистр или 128-разрядная ячейка памяти, в качестве выходного операнда должен выступать XMM-регистр. Схема работы команды показана на рис. 13.15.

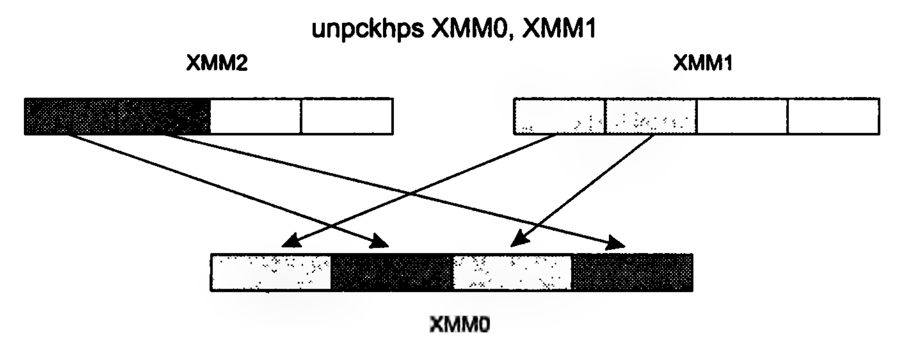



**Рис. 13.15.** Схема работы команды unpckhps

- unpcklps — параллельное перемещение младших двойных слов из операнда-источника и операнда-приемника в операнд-приемник. При этом два младших двойных слова операнда-источника становятся старшими двой-

ными словами в 64-разрядных элементах операнда-приемника, а два младших двойных слова операнда-приемника — младшими двойными словами в 64-разрядных элементах операнда-приемника. Входным операндом (источником) могут выступать XMM-регистр или 128-разрядная ячейка памяти, в качестве выходного операнда должен выступать XMM-регистр. Схема работы команды показана на рис. 13.16.

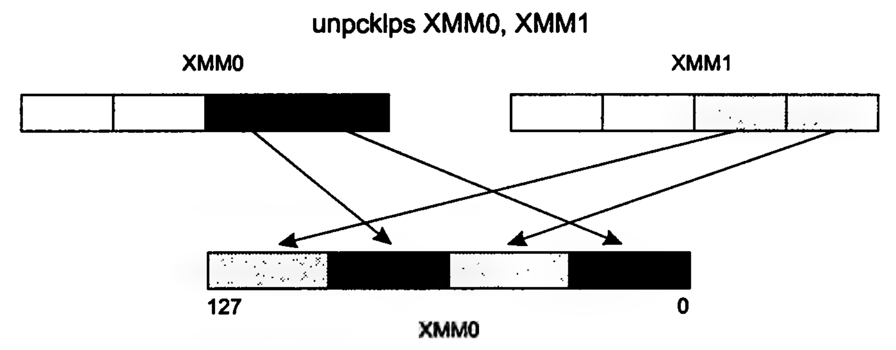



**Рис. 13.16.** Схема работы команды unpcklps

К командам распаковки данных можно отнести и команду shufps, выполняющую перестановку 32-разрядных упакованных операндов в соответствии с заданной маской. Команда имеет три операнда: входной, выходной и операнд-маску. Маска представляет собой непосредственное 8-разрядное значение, задающее порядок перестановки операндов. Каждая пара битов маски определяет номер упакованного 32-разрядного операнда в приемнике или источнике, который должен помещаться в операнд-приемник. При этом порядок размещения 32-разрядных операндов таков: младшие 4 бита маски указывают номера двух упакованных чисел приемника, которые становятся младшими упакованными значениями результата, а старшие 4 бита — номера упакованных чисел источника, которые становятся старшими упакованными значениями результата.

Должен заметить, что все перестановки выполняются одновременно, то есть параллельно. Лучше всего схему работы команды иллюстрирует рис. 13.17.

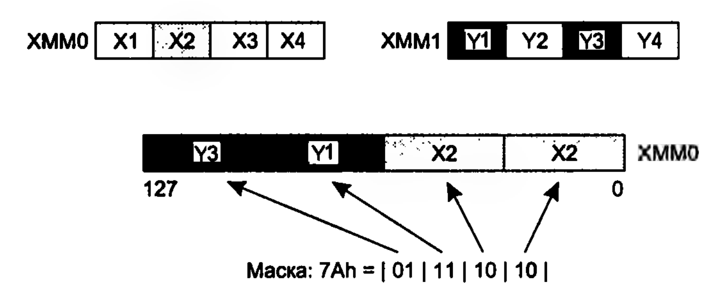



**Рис. 13.17.** Схема работы команды shufps

Как видно из рис. 13.17, младшие пары битов маски равны 10, то есть 2, поэтому оба младших операнда результата будут содержать элемент X2. Старшие пары битов маски равны 11 (3) и 01 (1), поэтому старшие элементы результата будут содержать Y1 и Y3.

Продемонстрирую работу команды shufps на примере простой процедуры (назовем ее _shufps_ex), исходный текст которой показан в листинге 13.23.

**Листинг 13.23.** Применение команды shufps

```
.686
.model flat
.XMM
option casemap:none
.data
  res DD 4 DUP (0)
.code
 _shufps_ex proc
  push   EBP
  mov    EBP, ESP
  mov    ESI, dword ptr [EBP+8]
  mov    EDI, dword ptr [EBP+12]
  lea    EBX, res
  movups XMM0, [ESI]
  movups XMM1, [EDI]
  shufps XMM0, XMM1, 7Ah
  movups [EBX], XMM0
  lea    EAX, res
  pop    EBP
  ret
 _shufps_ex endp
 end
```

Процедура принимает два параметра — адреса двух массивов чисел с плавающей точкой. Кроме того, для упрощения вычислений предположим, что оба массива содержат 4 элемента размером в двойное слово.

Параметры извлекаются при помощи регистра EBP, при этом адрес первого массива загружается в регистр ESI, адрес второго — в EDI, а адрес массива res, содержащего результат, — в регистр EBX.

После этого поместим первое 128-разрядное значение (4 двойных слова первого массива) в регистр XMM0 (команда movups XMM0, [ESI]), а второе 128-разрядное значение (4 элемента второго массива) в регистр XMM1 (команда movups XMM1, [EDI]). Наконец, выполним перестановку упакованных операндов, используя маску 7Ah, при помощи команды

```
shufps XMM0, XMM1, 7Ah
```

Полученный в регистре XMM0 результат поместим в массив res командой

```
movups [EBX], XMM0
```

Предпоследняя команда процедуры помещает адрес результата в регистр EAX:

```
lea EAX, res
```

После этого происходит выход из процедуры.

Чтобы посмотреть на результат работы процедуры _shufps_ex, воспользуемся консольной программой на Visual C++ .NET (листинг 13.24).

**Листинг 13.24.** Демонстрационная программа для процедуры из листинга 13.23

```
#include <stdio.h>
extern "C" float* shufps_ex(float* f1, float* f2);
int main(void)
{
 float f1[4] = { 1.11, 2.22, 3.33, 4.44};
 float f2[4] = { -1.11, -2.22, -3.33, -4.44 };
 float* pf = shufps_ex(f1, f2);
 printf("SHUFPS example:\n");
 for (int i1 = 0; i1 < 4; i1++)
  {
   printf("%5.2f ", *pf++);
  }
 return 0;
}
```

Если запустить программу, то на экран будет выведен такой результат:

```
SHUFPS example:
3.33 3.33 -4.44 -2.22
```

Этот результат соответствует той схеме, которая показана на рис. 13.17: элементы с номером 2 регистра XMM0 (а это 3,33) помещаются в младшие двойные слова результата, а элементы с номерами 3 (–4,44) и 1 (–2,22) регистра XMM1 — в старшие двойные слова результата, который оказывается в регистре XMM0.

Команду shufps можно задействовать для различных манипуляций упакованными элементами, причем с ее помощью можно реализовывать довольно серьезные алгоритмы. Вот простейший вариант такого использования команды shufps: пусть требуется поменять порядок следования элементов в 128-разрядном операнде, то есть сделать младший операнд старшим, а старший — младшим и т. д. Подобную манипуляцию можно выполнить с помощью несложной процедуры (назовем ее _reverse_ex).

Процедура принимает один параметр — адрес 128-разрядного операнда (или массива из четырех 32-разрядных чисел, что в данном случае одно и то же), а возвращает, как обычно, адрес массива с результатом в регистре EAX. Исходный текст процедуры _reverse_ex представлен в листинге 13.25.

**Листинг 13.25.** Изменение порядка следования элементов в массиве

```
.686
.model flat
.XMM
option casemap:none
.data
  res DD 4 DUP (0)
.code
 _reverse_ex proc
  push   EBP
  mov    EBP, ESP
  mov    ESI, dword ptr [EBP+8]
```

продолжение ⇗

**Листинг 13.25** (продолжение)

```
lea    EBX, res
movups XMM0, [ESI]
shufps XMM0, XMM0, 1Bh
movups [EBX], XMM0
lea    EAX, res
pop    EBP
ret
_reverse_ex endp
end
```

Это очень простая процедура, и я не буду детально ее анализировать. Хочу лишь обратить ваше внимание на то, как используется команда shufps: в качестве обоих операндов указан один и тот же регистр (в данном случае — XMM0), а маской является значение 1Bh.

Если, предположим, исходный массив содержит элементы в следующем порядке:

```
1.11 2.22 3.33 4.44
```

то после выполнения процедуры _reverse_ex результирующий массив будет содержать элементы, расположенные так:

```
4.44 3.33 2.22 1.11
```

На этом можно закончить обзор команд распаковки данных. Последняя группа команд, которую мы проанализируем, — команды управления кэшированием данных.

# 13.8. Команды управления кэшированием

Необходимость управления кэшированием вызвана тем, что большинство мультимедийных приложений оперируют большими объемами данных, при этом может случиться, что данные, загруженные в кэш, никогда не понадобятся. Чтобы оптимизировать работу кэша, в систему команд SSE-расширения и были включены команды управления кэшем. Вот их перечень:

- maskmovq — выборочное сохранение в памяти байтов упакованных данных MMX-регистра. В качестве операнда-источника используется один из MMX-регистров, а операндом-приемником служит область памяти, адрес которой задан в регистре EDI. Маска указывает, какие байты будут сохранены в памяти, и формируется из старших разрядов каждого байта, находящегося в MMX-регистре;
- movntq — запись в память, минуя кэш, целочисленных упакованных данных в формате MMX. Операндом-источником здесь выступает MMX-регистр, а операндом-приемником — 64-разрядная ячейка памяти;
- movnps — запись в память, минуя кэш, упакованных чисел с плавающей точкой в коротком формате. Операндом-источником здесь выступает XMM-регистр, а операндом-приемником — 128-разрядная ячейка памяти, адрес которой должен быть выровнен по 16-байтовой границе.

Мы закончили анализ аппаратно-программной архитектуры SSE-расширения и можем сделать некоторые выводы относительно этой технологии. Ее использование значительно ускоряет работу приложений при обработке больших объемов данных при ограниченных ресурсах времени, поскольку данные могут обрабатываться параллельно в одном цикле. Операции с упакованными числами с плавающей точкой обладают повышенной точностью, и при прочих равных условиях следует отдавать им предпочтение.

Прежде чем использовать широкие возможности, предоставляемые технологией SSE по оптимизации программ, следует тщательно продумать алгоритм задачи и оценить целесообразность применения этой технологии. Из-за ограниченности объема книги мы рассмотрели не все возможности SSE, но я надеюсь, что читатели извлекли немалую пользу для себя и смогут эффективно задействовать SSE-команды в своих программах.

# Технология SSE2 в процессорах Intel Pentium 4



Технология SSE2 (Streaming SIMD Extensions 2) разработана для применения в процессорах Intel Pentium 4. Ее назначение — повысить эффективность операций со 128-разрядными данными в формате плавающей точки с двойной точностью (double-precision floating point) и с целочисленными данными. Эта технология позволяет разрабатывать высокопроизводительные приложения для 3D-графики и 3D-геометрии, моделирования и симуляции процессов (и не только в математике), обработки сигналов, 3D-анимации, кодирования/декодирования, распознавания речи и т. д.

Технология SSE2 расширяет возможности MMX за счет использования 128-разрядных регистров вместо 64-разрядных, обеспечивая высокую эффективность параллельных вычислений. Достижение более высокой производительности возможно также за счет включения в SSE2 новых типов данных: 128-разрядных операндов с плавающей точкой двойной точности и 128-разрядных упакованных целых чисел. Технология SSE2 позволяет улучшить вычислительные возможности благодаря:

- улучшению управления данными в кэше;
- повышению производительности операций, требующих более высокой точности;
- расширению до 128 бит диапазона обрабатываемых 64-разрядными командами операндов.

Рассмотрим более подробно типы данных, которые использует SSE2-расширение. Основное преимущество SSE2 связано с применением 64-разрядных чисел с плавающей точкой двойной точности, формат которых показан на рис. 14.1.

SSE2-команды при выполнении операций используют восемь 128-разрядных регистров (XMM0 – XMM7) и могут работать в скалярном или параллельном режиме. SSE2-команды оперируют с такими типами данных, как:

- упакованные и скалярные числа с плавающей точкой в коротком формате;
- упакованные и скалярные числа с плавающей точкой двойной точности;
- упакованные и скалярные целые числа размером 128 бит.

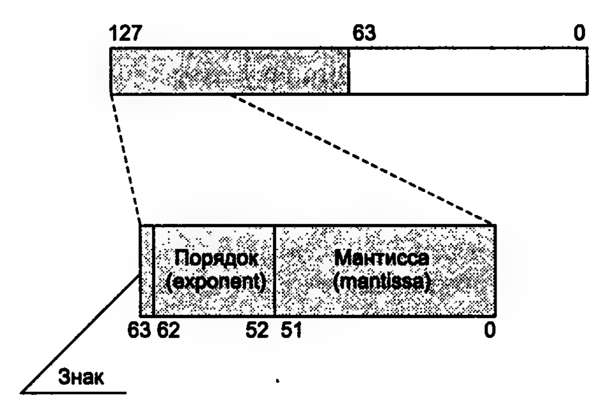



**Рис. 14.1.** Формат упакованного 64-разрядного числа с плавающей точкой двойной точности

Команды 128-разрядной целочисленной арифметики используют тот же набор регистров (XMM0 – XMM7), что и команды, оперирующие с числами с плавающей точкой. Инструкции SSE2-расширения не требуют применения команды emms, поскольку выполняются вне зависимости от сопроцессора. Кроме того, SSE2-команды позволяют:

- разрабатывать алгоритмы, в которых одновременно можно обрабатывать смешанные типы данных: упакованные числа с плавающей точкой в коротком формате и указанные с двойной точностью, а также целые 64- и 128-разрядные числа;
- работать с данными различной размерности: байтом, словом, двойным словом, учетверенным словом и двойным учетверенным словом.

Напомню, что с помощью параллельных команд можно одновременно обрабатывать все упакованные операнды, в то время как с помощью скалярных — только младший операнд. Команды SSE2-расширения в большинстве случаев требуют выравнивания адресов операндов в памяти по 16-байтовой границе, хотя из этого правила есть некоторые исключения, например команда загрузки, или сохранения, операнда в невыровненной области памяти. Еще один случай связан с использованием скалярной команды, работающей с переменной размером в 8 байт и не требующей выравнивания.

Мнемонические обозначения параллельных команд содержат суффикс pd, а скалярные — суффикс sd. Прежде чем приступить к анализу инструкций SSE2-расширения и практическим примерам, сделаю несколько уточнений. Для разработки ассемблерных программ, содержащих SSE2-команды, мы будем использовать компилятор MASM 7.10*xxxx* (включен в Windows XP DDK или Windows Server 2003 DDK). Исходный текст программы на ассемблере обязательно должен содержать директиву .XMM.

При разработке ассемблерных процедур с SSE2-командами в большинстве случаев требуется выравнивание адресов данных по 16-байтовой границе. С этим мы уже сталкивались при обсуждении SSE-расширения в главе 13, поэтому я не буду больше на этом останавливаться.

Перед тем как разрабатывать программный код с использованием технологии SSE2, необходимо убедиться в том, что процессор ее поддерживает. Для этого нужно, выполнив команду cpuid, проанализировать 26-й бит регистра EDX — если этот бит отличен от нуля, то технология SSE2 поддерживается процессором. Соответствующий фрагмент программного кода может быть таким:

```
. . .
SSE2supp DB 0
. . .
mov   EAX, 1
xor   EBX, EBX
cpuid
test  EDX, 4000000h ;анализ 26-го бита
setne BL
mov   SSE2support, BL
. . .
```

Если переменная SSE2support после выполнения этого фрагмента кода содержит 1, то технология SSE2 поддерживается, а если 0 — нет.

Приступим к анализу основных групп команд SSE2-расширения. Все команды в зависимости от типа обрабатываемых операндов можно разделить на две большие группы, предназначенные для обработки 128-разрядных операндов с плавающей точкой и 128-разрядных целочисленных операндов.

## 14.1. Команды обработки 128-разрядных данных с плавающей точкой

В группу команд обработки 128-разрядных данных с плавающей точкой входят следующие команды:

- перемещения (пересылки, передачи) данных;
- арифметические (сложения, вычитания, умножения, деления, извлечения квадратного корня и поиска максимума/минимума);
- сравнения;
- логических операций;
- распаковки и распределения данных;
- преобразования форматов данных;
- управления состоянием вычислений;
- управления кэшированием данных.

Первая группа команд, которые мы рассмотрим, — команды перемещения данных.

К командам перемещения данных относятся:

- movapd — пересылка 128-разрядных упакованных данных с плавающей точкой двойной точности из входного операнда (источника) в выходной операнд (приемник). В качестве источника и приемника могут выступать XMM-

регистр или 128-разрядная ячейка памяти, при этом хотя бы один из операндов должен быть XMM-регистром. Адрес операнда в памяти должен быть выровнен по 16-байтовой границе, в противном случае генерируется исключение общей защиты;

- movhpd — пересылка старших 64 битов XMM-регистра в память и наоборот. Данные пересылаются из 64-разрядной ячейки памяти в старшую часть XMM-регистра. В качестве операнда-источника и операнда-приемника могут выступать XMM-регистр или ячейка памяти. Если выполняется пересылка данных в XMM-регистр, то младшая часть регистра не изменяется;
- movlpd — пересылка младших 64 битов XMM-регистра в память и наоборот. Данные пересылаются из 64-разрядной ячейки памяти в младшую часть XMM-регистра. В качестве операнда-источника и операнда-приемника могут выступать XMM-регистр или ячейка памяти. Если выполняется пересылка данных в XMM-регистр, то старшая часть регистра не изменяется;
- movupd — пересылка невыровненых 128-разрядных упакованных данных с плавающей точкой двойной точности из входного операнда (источника) в выходной операнд (приемник). В качестве источника и приемника могут выступать XMM-регистр или 128-разрядная ячейка памяти, при этом хотя бы один из операндов должен быть XMM-регистром;
- movsd — пересылка скалярных данных размером 64 бит из младшей части XMM-регистра в память и наоборот. Данные пересылаются из 64-разрядной ячейки памяти в младшую часть XMM-регистра. В качестве операнда-источника и операнда-приемника могут выступать XMM-регистр или ячейка памяти. Если оба операнда являются XMM-регистрами, то пересылаются младшие части регистров. Если выполняется пересылка данных из памяти в XMM-регистр, то старшие 64 бита регистра устанавливаются в 0;
- movmskpd — сохранение знаковых битов каждого 64-разрядного операнда с плавающей точкой двойной точности в младших битах 32-разрядного регистра общего назначения. Это 2-разрядное значение может быть использовано для организации ветвлений в программе. Схема работы команды movmskpd показана на рис. 14.2;

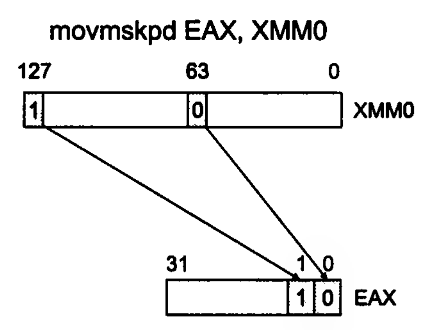
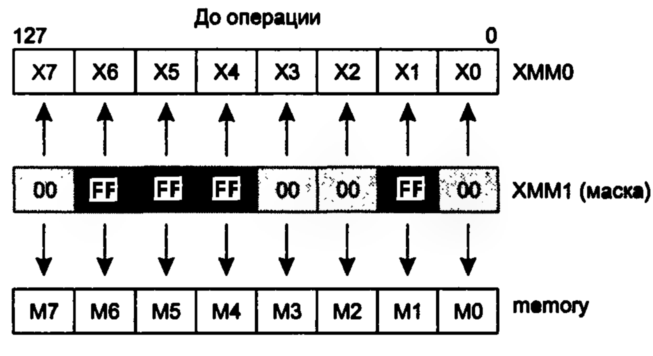



**Рис. 14.2.** Схема работы команды movmskpd

- maskmovdqu — пересылка содержимого операнда-источника в операнд-приемник в соответствии с указанной маской. Команда имеет три операнда: операнд-источник, операнд-приемник и маску. Операнд-источник и маска указываются явным образом в команде, а операнд-приемник находится по адресу, определяемому регистром EDI. Команда работает по следующей схеме: если знаковый бит элемента маски равен 1, то выполняется пересылка элемента-источника из соответствующей позиции в операнд-приемник на ту же позицию, а если знаковый бит равен 0, то пересылка не выполняется и соответствующий элемент в операнде-приемнике остается без изменений. Лучше представить себе работу команды maskmovdqu поможет рис. 14.3.

maskmovdqu XMM0, XMM1




После операции




**Рис. 14.3.** Схема работы команды maskmovdqu

Команда maskmovdqu позволяет создавать алгоритмы перестановки, сцепления и врезки элементов строк и массивов. Продемонстрирую это на примере врезки символов одной строки в другую. Программный код примера реализован в виде процедуры _maskmovdqu_ex (листинг 14.1).

**Листинг 14.1.** Применение команды maskmovdqu

```
.686
.model flat
.XMM
option casemap: none
.data
 res DB 16 DUP ('+').0
.code
 _maskmovdqu_ex proc
  push      EBP
  mov       EBP, ESP
  mov       ESI, dword ptr [EBP+8]
  mov       EBX, dword ptr [EBP+12]
```

```
  lea        EDI, res
  movdqu     XMM0, [ESI]
  movdqu     XMM1, [EBX]
  maskmovdqu XMM0, XMM1
  lea        EAX, res
  pop        EBP
  ret
 _maskmovdqu_ex endp
 end
```

Процедура принимает в качестве входных параметров адреса массива байтов и маски. Символы исходного массива копируются в массив res, содержащий символ +, если соответствующий этому символу байт маски равен шестнадцатеричному значению FF, в противном случае содержимое элемента массива res остается неизменным. Процедура возвращает адрес массива res в регистре EAX.

Для адресации исходного массива используется регистр ESI, маска адресуется регистром EBX, а операнд в памяти (массив res) — регистром EDI.

Для проверки работы процедуры можно использовать простое консольное приложение на Visual C++ .NET (листинг 14.2).

**Листинг 14.2.** Демонстрационная программа для процедуры из листинга 14.1

```
#include <stdio.h>
extern "C" unsigned char* maskmovdqu_ex(
  unsigned char* a1, unsigned char* msk);
int main(void)
{
 __declspec(align(16))unsigned char a1[] = "0123456789ABCDEF";
 __declspec(align(16))unsigned char msk[] = {
    0x0, 0xFF, 0x0, 0xFF, 0xFF, 0x0, 0xFF, 0xFF,
    0xFF,0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF };
 printf("MASKMOVDQU example:\n");
 printf("Source      string: %s\n", a1);
unsigned char* pa1 = maskmovdqu_ex(a1, msk);
printf("Destination string: %s\n", pa1);
return 0;
}
```

Здесь процедура maskmovdqu_ex объявлена как внешняя, принимающая в качестве параметра адреса массива беззнаковых символов (unsigned char* a1) и маски (unsigned char* msk) и возвращающая адрес массива результата.

При указанных значениях элементов массивов a1 и msk программа выводит на экран следующий результат:

```
MASKMOVDQU example:
Source string: 0123456789ABCDEF
Destination string: +1+34+6789ABCDEF
```

В группу арифметических команд входят команды сложения, вычитания, умножения и деления. К этой группе очень часто относят и команды извлечения квадратного корня, поиска максимального и минимального элементов. Команды этой группы могут работать как с упакованными операндами (параллельные команды), так и с обычными (скалярные команды). Вначале рассмотрим команды параллельного сложения и вычитания:

- addpd — параллельное сложение двух упакованных 64-разрядных чисел с плавающей точкой двойной точности. Команда принимает два операнда: в качестве входного операнда могут выступать XMM-регистр или ячейка памяти, выходным операндом может быть только XMM-регистр;
- subpd — параллельное вычитание двух упакованных 64-разрядных чисел с плавающей точкой двойной точности. Команда принимает два операнда: в качестве входного операнда могут выступать XMM-регистр или ячейка памяти, выходным операндом может быть только XMM-регистр. Команда вычитает содержимое операнда-источника из содержимого операнда-приемника и сохраняет результат в операнде-приемнике.

Следующие две команды работают со скалярными данными, оперируя младшими элементами 128-разрядных значений, не затрагивая старшие:

- addsd — сложение младших 64-разрядных операндов с плавающей точкой двойной точности. Команда принимает два операнда: в качестве входного операнда могут выступать XMM-регистр или ячейка памяти, выходным операндом может быть только XMM-регистр;
- subsd — вычитание младших 64-разрядных операндов с плавающей точкой двойной точности. Команда принимает два операнда: в качестве входного операнда могут выступать XMM-регистр или ячейка памяти, выходным операндом может быть только XMM-регистр. Команда вычитает содержимое операнда-источника из содержимого операнда-приемника и сохраняет результат в операнде-приемнике.

Приведу пример использования одной из команд — команды параллельного сложения addpd. Соответствующий программный код, реализованный в виде процедуры addpd_ex, позволяет вычислить сумму двух упакованных 128-разрядных операндов, каждый из которых представляет собой 64-разрядное число с плавающей точкой двойной точности (листинг 14.3). Процедуре передаются адреса операндов, а возвращаемым результатом является адрес области памяти, содержащей разность чисел.

**Листинг 14.3.** Вычисление суммы 128-разрядных операндов в формате плавающей точки двойной точности

```
.686
.model flat
option casemap: none
.XMM
.data
  res DQ 2 DUP (0)
.code
 addpd_ex proc
   push   EBP
   mov    EBP, ESP
   mov    ESI, dword ptr [EBP+8]
   mov    EDI, dword ptr [EBP+12]
   lea    EBX, res
   movupd XMM0, [ESI]
   addpd  XMM0, [EDI]
```

```
   movupd [EBX]. XMM0
   lea    EAX. res
   pop    EBP
   ret
 addpd_ex endp
 end
```

Следующий пример демонстрирует скалярное вычитание двух 64-разрядных чисел с плавающей точкой при помощи команды subsd. Программный код представляет собой процедуру (она называется subsd_ex), входными параметрами которой являются значения операндов (листинг 14.4). Процедура возвращает в регистре EAX адрес области памяти res, содержащей результат вычитания. Мнемонически процедуру можно представить так:

subsd_ex(*a1*. *a2*)

Здесь *a1*, *a2* — входные параметры. Тогда результат выполнения процедуры выглядит как *a1* – *a2*.

**Листинг 14.4.** Скалярное вычитание чисел с плавающей точкой двойного формата

```
.686
.model flat
option casemap: none
.XMM
.data
  res DQ 0
.code
 subsd_ex proc
   push   EBP
   mov    EBP. ESP
   lea    EBX. res
   movsd  XMM0. [EBP+8]
   subsd  XMM0. [EBP+16]
   movsd  [EBX]. XMM0
   lea    EAX. res
   pop    EBP
   ret
 subsd_ex endp
 end
```

Программный код процедуры несложен, но я хочу обратить внимание читателей на то, что параметры процедуры являются 8-байтовыми числами, поэтому второй параметр отстоит от первого на 8 байт и адресуется как [EBP+16].

Подгруппа команд умножения включает две команды:

- mulpd — параллельное умножение двух упакованных 64-разрядных чисел с плавающей точкой двойной точности. Команда принимает два операнда: в качестве входного операнда могут выступать XMM-регистр или ячейка памяти, выходным операндом может быть только XMM-регистр;
- mulsd — скалярное умножение младших 64-разрядных операндов с плавающей точкой двойной точности. Команда принимает два операнда: в качестве входного операнда могут выступать XMM-регистр или ячейка памяти, выходным операндом может быть только XMM-регистр. Команда не затрагивает старшие части операндов.

Подгруппа команд деления включает две команды:

- divpd — параллельное деление двух упакованных 64-разрядных чисел с плавающей точкой двойной точности. Команда принимает два операнда: в качестве входного операнда могут выступать XMM-регистр или ячейка памяти, выходным операндом может быть только XMM-регистр. Команда выполняет деление содержимого операнда-приемника на содержимое операнда-источника и сохраняет результат в операнде-приемнике;
- divsd — скалярное деление младших 64-разрядных операндов с плавающей точкой двойной точности. Команда принимает два операнда: в качестве входного операнда могут выступать XMM-регистр или ячейка памяти, выходным операндом может быть только XMM-регистр. Команда выполняет деление содержимого операнда-приемника на содержимое операнда-источника и сохраняет результат в операнде-приемнике, не затрагивая старшие части операндов.

Приведу пример процедуры, в которой вычисляется значение выражения

$$\frac{a1-a2}{a1+a2},$$

где $a1$ и $a2$ — входные параметры процедуры (64-разрядные числа с плавающей точкой в двойном формате). Адрес результата возвращается в регистре EAX. Процедура называется _combo_ex, и в ней используются скалярные команды addsd, subsd и divsd. Исходный текст процедуры представлен в листинге 14.5.

**Листинг 14.5.** Вычисление выражения (a1 – a2)/(a1 + a2)

```
.686
.model flat
option casemap: none
.XMM
.data
  res DQ 0
.code
 _combo_ex proc
   push   EBP
   mov    EBP, ESP
   lea    EBX, res
   movsd  XMM0, [EBP+8]  ; a1 -> XMM0
   movsd  XMM1, XMM0     ; a1 -> XMM1
   subsd  XMM0, [EBP+16] ; a1-b1 -> XMM0
   addsd  XMM1, [EBP+16] ; a1+b1 -> XMM1
   divsd  XMM0, XMM1     ; (a1-b1)/(a1+b1) -> XMM0
   movsd  [EBX], XMM0
   lea    EAX, res
   pop    EBP
   ret
 _combo_ex endp
 end
```

К группе арифметических команд относят также команды извлечения квадратного корня. Существует две формы команды: sqrtpd (параллельная обработка) и sqrtsd (скалярная обработка):

- sqrtpd — параллельное извлечение квадратного корня из двух упакованных 64-разрядных чисел с плавающей точкой двойной точности. Команда принимает два операнда: в качестве входного операнда могут выступать XMM-регистр или ячейка памяти, выходным операндом может быть только XMM-регистр;
- sqrtsd — скалярное извлечение квадратного корня из младших 64-разрядных операндов с плавающей точкой двойной точности. Команда принимает два операнда: в качестве входного операнда могут выступать XMM-регистр или ячейка памяти, выходным операндом может быть только XMM-регистр. Старшие операнды во время выполнения не изменяются.

Поиск максимальных/минимальных значений выполняется при помощи команд maxpd/minpd (параллельная обработка) и maxsd/minsd (скалярная обработка):

- maxpd — параллельный поиск максимального значения в парах упакованных 64-разрядных чисел с плавающей точкой двойной точности. Команда принимает два операнда: в качестве входного операнда могут выступать XMM-регистр или ячейка памяти, выходным операндом может быть только XMM-регистр. Результат сохраняется в выходном операнде;
- minpd — параллельный поиск минимального значения в парах упакованных 64-разрядных чисел с плавающей точкой двойной точности. Команда принимает два операнда: в качестве входного операнда могут выступать XMM-регистр или ячейка памяти, выходным операндом может быть только XMM-регистр. Результат сохраняется в выходном операнде;
- maxsd — поиск максимального значения в паре младших 64-разрядных операндов с плавающей точкой двойной точности. Команда принимает два операнда: в качестве входного операнда могут выступать XMM-регистр или ячейка памяти, выходным операндом может быть только XMM-регистр. Результат сохраняется в младшей части выходного операнда, не затрагивая старший операнд;
- minds — поиск минимального значения в паре младших 64-разрядных операндов с плавающей точкой двойной точности. Команда принимает два операнда: в качестве входного операнда может выступать XMM-регистр или ячейка памяти, выходным операндом может быть только XMM-регистр. Результат сохраняется в младшей части выходного операнда, не затрагивая старший операнд.

В листинге 14.6 приводится пример параллельного поиска минимальных значений среди пар 64-разрядных операндов с плавающей точкой двойной точности. Программный код реализован в виде процедуры _minpd_ex, принимающей

в качестве параметров адреса 128-разрядных операндов и возвращающей адрес результата в регистре EAX.

**Листинг 14.6.** Параллельный поиск минимальных значений среди пар элементов

```
.686
.model flat
option casemap: none
.XMM
.data
  res DQ 0
.code
 _minpd_ex proc
   push   EBP
   mov    EBP, ESP
   mov    ESI, dword ptr [EBP+8]
   mov    EDI, dword ptr [EBP+12]
   lea    EBX, res
   movupd XMM0, [ESI]
   minpd  XMM0, [EDI]
   movupd [EBX], XMM0
   lea    EAX, res
   pop    EBP
   ret
 _minpd_ex endp
 end
```

Для проверки работы процедуры можно использовать простую программу, написанную на Visual C++ .NET (листинг 14.7).

**Листинг 14.7.** Демонстрационная программа для процедуры из листинга 14.6

```
#include <stdio.h>
extern "C" double* minpd_ex(double* a1, double* a2);
int main(void)
{
 __declspec(align(16))double a1[2] = { 345.98, 274.16 };
 __declspec(align(16))double a2[2] = { 562.49, -712.73 };
 double* pa = minpd_ex(a1, a2);
 printf("MINPD example:\n");
 for (int i1 = 0;i1 < 2; i1++)
 {
  printf("%5.2f ", *pa++);
 }
 return 0;
}
```

Здесь при помощи директивы extern процедура minpd_ex объявлена внешней. Кроме того, и об этом уже упоминалось, SSE2-команды требуют, чтобы адреса переменных были выровнены по 16-байтовой границе. Обратите внимание на то, что 64-разрядным числам с плавающей точкой в двойном формате в языке Visual C++ .NET соответствует тип double. При указанных значениях элементов массивов a1 и a2 программа выводит на экран следующий результат:

```
MINPD example:
345.98 -712.73
```

В группу команд сравнения входят следующие команды:

- cmpxxxpd — параллельное сравнение двух пар упакованных 64-разрядных чисел с плавающей точкой двойной точности. По результату сравнения в операнд-приемник записывается 64-разрядная маска в соответствующей позиции, состоящая либо из единиц, либо из нулей, в зависимости от результата сравнения. В качестве входного операнда выступают XMM-регистр или ячейка памяти, выходным операндом может быть только XMM-регистр. Для этой команды поддерживается набор условий, который условно обозначен символами *xxx* (табл. 14.1).

**Таблица 14.1.** Набор условий для команды cmpxxxpd

| Код условия | Описание | Условие |
|---|---|---|
| eq | Equal | Равно |
| lt | Less-than | Меньше чем |
| le | Less-than-or-equal | Меньше чем или равно |
| unord | Unordered | Неупорядоченный операнд |
| neq | Not-equal | Не равно |
| nlt | Not-less-than | Не меньше чем |
| nle | Not-less-than-or-equal | Не меньше чем или равно |
| ord | Ordered | Упорядоченный операнд |

Например, следующая команда выполняет параллельное сравнение на равенство пар упакованных 64-разрядных операндов в регистрах XMM0 и XMM1:

```
cmpeqpd XMM0, XMM1
```

- cmpxxxsd — скалярное сравнение младших частей 128-разрядных операндов. По результату сравнения в младшей части возвращается 64-разрядная маска, состоящая либо из единиц, либо из нулей, в зависимости от результата сравнения. В качестве входного операнда выступают XMM-регистр или ячейка памяти, выходным операндом может быть только XMM-регистр. Старшая часть операнда-приемника во время выполнения операции не изменяется. Для этой команды также поддерживается набор условий *xxx* (см. табл. 14.1), которые можно включить в мнемонику команды. Например, следующая команда при сравнении младших 64-разрядных операндов, находящихся в регистрах XMM0 и XMM1, проверяет условие XMM0 < XMM1:

```
cmpltsd XMM0, XMM1
```

В группу команд сравнения входят еще две скалярные команды: comisd и ucomisd. Команды имеют два операнда и выполняют скалярное сравнение младших 64-разрядных чисел с плавающей точкой. После выполнения этих команд содержимое обоих операндов остается неизменным, но в регистре флагов EFLAGS процессора определенным образом устанавливаются флаги ZF, PF и CF, а флаги OF, SF и AF сбрасываются в 0. В качестве входных операндов обеих команд могут выступать XMM-регистры или 64-разрядные переменные в памяти, выходными операндами могут быть только XMM-регистры.

Различие команд ucomisd и comisd состоит в том, что они генерируют исключительные ситуации для различных форматов не-чисел (NAN). Эти команды очень удобны при организации ветвлений в программах, поскольку по состоянию флагов позволяют интерпретировать результат сравнения. В табл. 14.2 приводится соответствие между сравниваемыми операндами и устанавливаемыми флагами.

**Таблица 14.2.** Результат операций сравнения и состояние флагов

| Результат сравнения операндов op1 и op2 | Флаг ZF | Флаг PF | Флаг CF |
|---|---|---|---|
| Операнды неупорядочены (unordered) | 1 | 1 | 1 |
| op1 < op2 | 0 | 0 | 1 |
| op1 > op2 | 0 | 0 | 0 |
| op1 = op2 | 1 | 0 | 0 |

В группу команд распаковки и перестановки входят две команды:

- unpckhpd — параллельное перемещение старших 64-разрядных чисел с плавающей точкой из операнда-источника и операнда-приемника в операнд-приемник. При этом старший 64-разрядный элемент операнда-источника становится старшим 64-разрядным элементом операнда-приемника, а старший 64-разрядный элемент операнда-приемника — младшим 64-разрядным элементом операнда-источника. Входным операндом (источником) могут быть XMM-регистр или 128-разрядная ячейка памяти, в качестве выходного операнда должен выступать XMM-регистр. Схема работы команды unpckhpd показана на рис. 14.4;

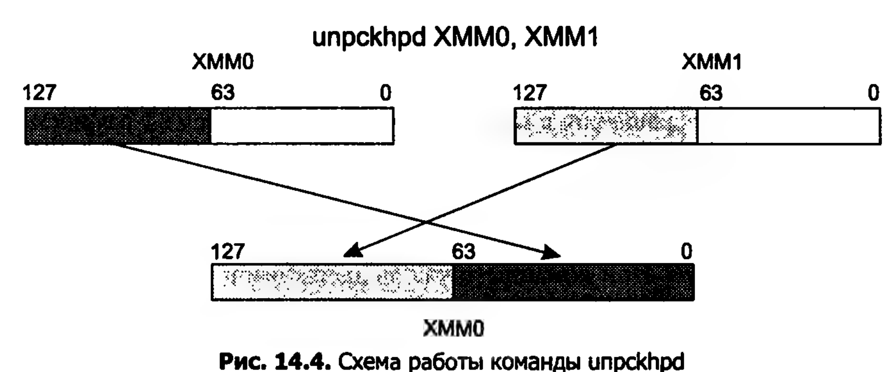



**Рис. 14.4.** Схема работы команды unpckhpd

- unpcklpd — параллельное перемещение младших 64-разрядных чисел с плавающей точкой из операнда-источника и операнда-приемника в операнд-приемник. При этом младший 64-разрядный элемент операнда-источника становится старшим 64-разрядным элементом операнда-приемника, а младший 64-разрядный элемент операнда-приемника — младшим 64-разрядным элементом операнда-приемника. Входным операндом (источником) могут быть XMM-регистр или 128-разрядная ячейка памяти, в качестве выходного операнда должен выступать XMM-регистр. Схема работы команды unpcklpd показана на рис. 14.5.

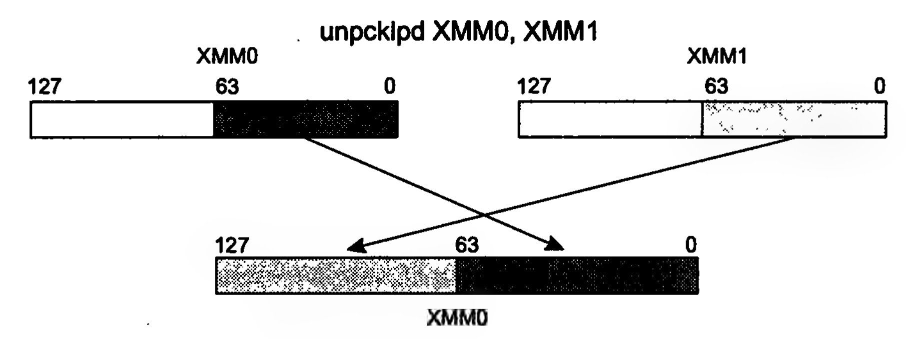



**Рис. 14.5.** Схема работы команды unpcklpd

К командам распаковки данных можно отнести и команду shufpd, выполняющую перестановку 64-разрядных упакованных чисел с плавающей точкой двойной точности в соответствии с заданной маской. Команда имеет три операнда: входной, выходной и маску. Маска представляет собой 8-разрядное значение, задающее порядок перестановки операндов. Младшие 2 бита маски определяют номер упакованного 64-разрядного операнда в приемнике или источнике, который должен помещаться в операнд-приемник. При этом порядок размещения 64-разрядных операндов таков: нулевой (младший) бит маски указывает номер упакованного числа приемника, которое становится младшим упакованным значением результата, а первый бит — номер упакованного числа источника, которое становится старшим упакованным значением результата.

В качестве входного операнда (источника) могут выступать XMM-регистр или 128-разрядная ячейка памяти. Выходным операндом может быть только XMM-регистр.

Должен заметить, что все перестановки выполняются одновременно, то есть параллельно. Лучше всего схему работы команды shufpd иллюстрирует рис. 14.6.

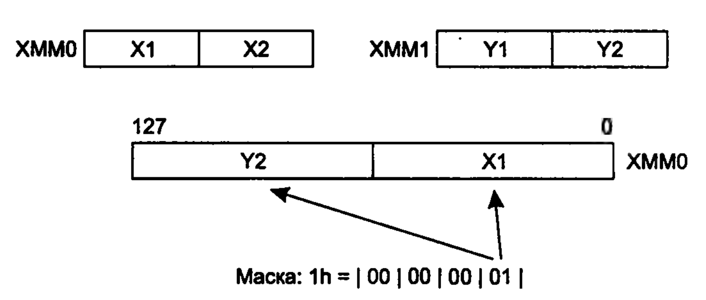



**Рис. 14.6.** Схема работы команды shufpd

Как видно из рис. 14.6, младший (нулевой) бит маски равен 1, поэтому младший операнд результата (регистр XMM0) будет содержать элемент *X*1. Первый бит маски равен 0, поэтому старший элемент результата будет содержать *Y*2.

Команда shufpd имеет ценное практическое применение. Например, с ее помощью можно поместить одно и то же 64-разрядное число в старшую и младшую части 128-разрядного операнда. Листинг 14.8 демонстрирует это.

**Листинг 14.8.** Использование команды shufpd

```
.686
.model flat
.XMM
option casemap:none
.data
  res DQ 2 DUP(0)
.code
shufpd_ex proc
  push   EBP
  mov    EBP, ESP
  mov    ESI, dword ptr [EBP+8]
  lea    EDI, res
  movupd XMM0, [ESI]
  shufpd XMM0, XMM0, 3h
  movupd [EDI], XMM0
  lea    EAX, res
  pop    EBP
  ret
shufpd_ex endp
end
```

Программный код примера реализован в виде процедуры shufpd_ex, принимающей в качестве параметра адрес 128-разрядного операнда. Процедура возвращает адрес 128-разрядного операнда, содержащего результат, в регистре EAX. Следующая команда программного кода процедуры помещает в оба 64-разрядных операнда регистра XMM0 значение старшего операнда:

```
shufpd XMM0, XMM0, 3h
```

Это можно проиллюстрировать схемой, показанной на рис. 14.7.

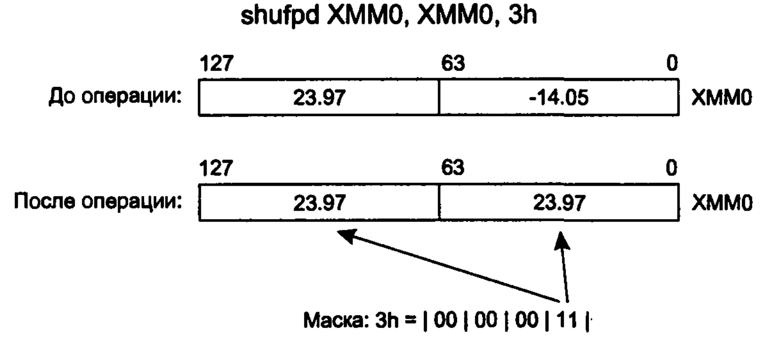



**Рис. 14.7.** Схема работы команды shufpd XMM0, XMM0, 3h

Если оба 64-разрядных операнда должны содержать значение младшей части, то в команде shufpd следует изменить значение маски с 3h на 0h.

Для того чтобы поменять обе 64-разрядные части регистра XMM0 местами, то есть старшую часть поместить на место младшей, а младшую — на место старшей, следует использовать команду

```
shufpd XMM0. XMM0. 1h
```

Команды преобразования обеспечивают преобразование упакованных или скалярных чисел с плавающей точкой в коротком формате (SPFP) или в двойном формате (DPFP) в формат целых чисел и наоборот. Данные в целочисленном формате могут быть 128-, 64- или 32-разрядными. К этой группе относятся следующие команды:

- cvtpd2pi — параллельное преобразование двух упакованных 64-разрядных чисел с плавающей точкой в двойном формате в два 32-разрядных целых числа со знаком. В качестве входного операнда (источника) может выступать один из XMM-регистров, а в качестве выходного операнда — MMX-регистр. Результат округляется в соответствии со значением битов поля **rc** регистра управления/состояния (MXCSR);
- cvttpd2pi — параллельное преобразование двух упакованных 64-разрядных чисел с плавающей точкой в двойном формате в два 32-разрядных целых числа со знаком. В качестве входного операнда (источника) может выступать один из XMM-регистров, а в качестве выходного операнда — MMX-регистр. Результат формируется путем отбрасывания дробной части числа, при этом регистр управления/состояния (MXCSR) не используется;
- cvtsd2si — скалярное преобразование младшего упакованного 64-разрядного числа с плавающей точкой в двойном формате в 32-разрядное целое число со знаком. В качестве входного операнда (источника) может выступать один из XMM-регистров, а в качестве выходного операнда — 32-разрядный регистр общего назначения. Результат округляется в соответствии со значением битов поля **rc** регистра управления/состояния (MXCSR);
- cvttsd2si — скалярное преобразование младшего упакованного 64-разрядного числа с плавающей точкой в двойном формате в 32-разрядное целое число со знаком. В качестве входного операнда (источника) может выступать один из XMM-регистров, а в качестве выходного операнда — 32-разрядный регистр общего назначения. Результат формируется путем отбрасывания дробной части числа, при этом регистр управления/состояния (MXCSR) не используется;
- cvtpi2pd — параллельное преобразование двух упакованных 32-разрядных целых чисел в два упакованных 64-разрядных числа с плавающей точкой. В качестве операнда-источника может выступать MMX-регистр, а в качестве операнда-приемника — XMM-регистр;
- cvtsi2sd — скалярное преобразование 32-разрядного целого числа в упакованное 64-разрядное число с плавающей точкой. Результат помещается в младшую часть XMM-регистра, не затрагивая старшей части. В качестве операнда-источника может выступать 32-разрядный регистр общего назначения, а в качестве операнда-приемника — XMM-регистр.

Рассмотрим практические аспекты использования команд преобразования и начнем с команды cvtpd2pi. Следующий пример показывает, как можно преобразовать два упакованных 64-разрядных числа с плавающей точкой в двойном формате в 32-разрядные упакованные целые числа (листинг 14.9). Программный код примера реализован в виде процедуры _cvtpd2pi_ex, принимающей в качестве параметра адрес 128-разрядного операнда и возвращающей в регистре EAX адрес области памяти res, содержащей два 32-разрядных целых числа.

**Листинг 14.9.** Преобразование чисел с плавающей точкой в двойном формате в целые числа

```
.686
.model flat
.XMM
option casemap: none
.data
 res DD 2 DUP(0)
.code
 _cvtpd2pi_ex proc
   push     EBP
   mov      EBP, ESP
   mov      ESI, dword ptr [EBP+8]
   movups   XMM0, [ESI]
   cvtpd2pi MM0, XMM0
   movq     qword ptr res, MM0
   lea      EAX, res
   pop      EBP
   ret
  _cvtpd2pi_ex endp
  end
```

Программный код процедуры несложен. Параметр (адрес 128-разрядной переменной), как обычно, передается через регистр EBP и загружается в ESI. Затем два 64-разрядных операнда с плавающей точкой помещаются в регистр XMM0 и выполняется их преобразование к целочисленному типу при помощи команды

```
cvtpd2pi MM0, XMM0
```

Результат преобразования, сохраненный в MMX-регистре MM0 при помощи команды movq, помещается в переменную res. Адрес этой переменной помещается в регистр EAX командой

```
lea EAX, res
```

После этого стек восстанавливается и происходит выход из процедуры.

По умолчанию режимом округления, определяемым полем **rc** регистра управления/состояния (MXCSR), является округление к ближайшему целому числу.

Для проверки работоспособности процедуры можно использовать короткую программу на Visual C++ .NET (листинг 14.10).

**Листинг 14.10.** Демонстрационная программа для процедуры из листинга 14.9

```
#include <stdio.h>
extern "C" int* cvtpd2pi_ex(double* a1);
int main(void)
```

```
{
__declspec(align(16)) double a1[2] = { -1956.54, 6720.47 };
int* pi1 = cvtpd2pi_ex(a1);
printf("CVTPD2PI example: \n");
printf("%d ", *pi1++);
printf("%d\n", *pi1);
return 0;
}
```

При указанных значениях элементов массива a1 чисел с плавающей точкой двойной точности полученный результат будет выглядеть так:

```
CVTPD2PI example:
-1957 6720
```

Следующий пример демонстрирует работу скалярной команды cvttsd2si. Напомню, что эта команда преобразует младшее 64-разрядное число с плавающей точкой двойной точности XMM-регистра в 32-разрядное целое число без дробной части, которое помещается в один из 32-разрядных регистров общего назначения. Программный код примера реализован в виде процедуры _cvttsd2si_ex (листинг 14.11). Процедура преобразует элементы массива чисел с плавающей точкой двойной точности в 32-разрядные целые числа. В качестве параметров процедура принимает (слева направо) адрес исходного массива и размер массива в байтах. Процедура возвращает адрес массива res целых чисел в регистре EAX.

**Листинг 14.11.** Скалярное преобразование чисел с плавающей точкой в целые числа

```
.686
.model flat
option casemap: none
.XMM
.data
  res DD 32 DUP(0)
.code
 _cvttsd2si_ex proc
  push      EBP
  mov       EBP, ESP
  mov       ESI, dword ptr [EBP+8]
  lea       EDI, res
  mov       ECX, dword ptr [EBP+12]
  shr       ECX, 3
next:
  movsd     XMM0, [ESI]
  cvttsd2si EAX, XMM0
  mov       [EDI], EAX
  add       ESI, 8
  add       EDI, 4
  dec       ECX
  jnz       next
  lea       EAX, res
  pop       EBP
  ret
_cvttsd2si_ex endp
 end
```

Параметры в процедуру передаются через регистр EBP, при этом адрес исходного массива помещается в регистр ESI, а размер этого массива (в байтах) — в регистр ECX. Далее размер в байтах преобразуется к количеству учетверенных (64-разрядных) слов при помощи команды

```
shr ECX. 3
```

Адрес массива res, где будет находиться результат преобразования, загружается в регистр EDI.

Все последующие вычисления выполняются в цикле next, счетчик которого находится в регистре ECX. Первая команда цикла помещает 64-разрядный операнд в младшую часть регистра XMM0:

```
movsd XMM0. [ESI]
```

Собственно преобразование выполняет команда

```
cvttsd2si EAX. XMM0
```

Эта команда помещает 32-разрядный целочисленный результат в регистр EAX (можно использовать и другой 32-разрядный регистр общего назначения). Следующие две команды продвигают указатели адресов исходного и результирующего массивов к следующим элементам:

```
add  ESI. 8
add  EDI. 4
```

Здесь необходимо учитывать то, что следующий 64-разрядный элемент исходного массива, адресуемый регистром ESI, имеет смещение 8 по отношению к текущему, а следующий элемент целочисленного массива res (адресуется регистром EDI) — смещение 4.

После окончания цикла адрес массива с результатами вычислений помещается в регистр EAX, после чего стек восстанавливается и происходит выход из процедуры.

На результат преобразования не влияет содержимое поля **rc** регистра управления/состояния (MXCSR). Проверить работу процедуры _cvttsd2si можно с помощью программы на Visual C++ .NET (листинг 14.12).

**Листинг 14.12.** Демонстрационная программа для процедуры из листинга 14.11

```
#include <stdio.h>
extern "C" int* cvttsd2si_ex(double* a1. int asize):
int main(void)
{
__declspec(align(16)) double a1[5] = {
     -16.54.72.47. -3774.03.45.98. -65.51 }:
int asize = sizeof(a1):
int* pi1 = cvttsd2si_ex(a1. asize):
printf("CVTTSD2SI example: \n"):
for (int i1 = 0:i1 < asize / 8: i1++)
 {
  printf("%d ". *pi2++):
 }
return 0:
}
```

При указанных значениях элементов массива a1 чисел с плавающей точкой двойной точности полученный результат будет выглядеть так:

```
CVTTSD2SI example:
-16 72 -3774 45 -65
```

Последний пример применения команд преобразования, который мы рассмотрим, связан с изменением режима округления. Предположим, что в процессе преобразования 64-разрядных чисел с плавающей точкой двойного формата в целочисленное значение требуется округлять результат к ближайшему большему числу. В этом случае перед выполнением операции преобразования следует установить биты поля **rc** регистра управления/состояния (MXCSR) в 10. Модифицируем исходный текст представленной в листинге 14.9 процедуры _cvtpd2pi_ex так, чтобы округление выполнялось в большую сторону. Назовем новую процедуру _cvtpd2pi_exgt (листинг 14.13).

**Листинг 14.13.** Округление числа с плавающей точкой в сторону большего целого числа

```
.686
.model flat
.XMM
option casemap: none
.data
  state_MXCSR DD 0
  res DD 2 DUP(0)
.code
 _cvtpd2pi_exgt proc
   push    EBP
   mov     EBP, ESP
   mov     ESI, dword ptr [EBP+8]
   stmxcsr state_MXCSR
   or      word ptr state_MXCSR, 4000h
   ldmxcsr state_MXCSR
   movups  XMM0, [ESI]
   cvtpd2pi MM0, XMM0
   movq    qword ptr res, MM0
   lea     EAX, res
   pop     EBP
   ret
  _cvtpd2pi_exgt endp
  end
```

Я не буду анализировать весь программный код процедуры, поскольку подобный код мы уже рассматривали. Остановлюсь лишь на изменениях. Для того чтобы установить режим округления к большему числу, в программный код процедуры добавлены три команды:

```
stmxcsr  state_MXCSR
or          word ptr state_MXCSR, 4000h
ldmxcsr  state_MXCSR
```

Первая из этих команд сохраняет состояние регистра MXCSR в переменной state_MXCSR. Все значащие биты состояния находятся в младшем слове, в частности

поле **rc** определяется битами 13–14. Для установки режима округления в большую сторону (**rc** = 10) выполняется команда

```
or  word ptr state_MXCSR, 4000h
```

После этого содержимое переменной state_MXCSR записывается обратно в регистр MXCSR командой

```
ldmxcsr state_MXCSR
```

Эти команды следует выполнить перед операцией преобразования, чтобы изменения возымели эффект.

Если запустить тестирующую программу, то результатом преобразования чисел с плавающей точкой двойной точности (–1956,54 и 6720,47) будут целые числа (–1956 и 6721).

Логические команды выполняют поразрядные операции над упакованными операндами:

- andpd — поразрядное логическое И над двумя операндами, каждый из которых представляет собой 128-разрядное значение. В качестве входного операнда (источника) могут выступать XMM-регистр или 128-разрядная ячейка памяти, а в качестве выходного — XMM-регистр;
- andnpd — инвертирует содержимое операнда-приемника, после чего выполняет операцию поразрядного логического И над двумя операндами, каждый из которых представляет собой 128-разрядное значение. В качестве входного операнда (источника) могут выступать XMM-регистр или 128-разрядная ячейка памяти, а в качестве выходного — XMM-регистр;
- orpd — поразрядное логическое ИЛИ над двумя операндами, каждый из которых представляет собой 128-разрядное значение. В качестве входного операнда (источника) могут выступать XMM-регистр или 128-разрядная ячейка памяти, а в качестве выходного — XMM-регистр;
- xorpd — поразрядное логическое исключающее ИЛИ над двумя операндами, каждый из которых представляет собой 128-разрядное значение. В качестве входного операнда (источника) могут выступать XMM-регистр или 128-разрядная ячейка памяти, а в качестве выходного — XMM-регистр.

Логические команды можно использовать для вычисления абсолютной величины (модуля) числа и изменения знака числа. Приведу два примера такого применения логических команд.

Для вычисления абсолютной величины упакованных 64-разрядных чисел с плавающей точкой двойной точности можно воспользоваться процедурой abs_ex, принимающей в качестве единственного параметра адрес 128-разрядного операнда, а возвращающей в регистре EAX адрес области памяти, содержащей результат (листинг 14.14).

Остановлюсь на программном коде процедуры более подробно. Как известно, знак числа определяется его старшим разрядом. Если число положительно, то старший или, как его называют, знаковый бит равен 0. Знаковый бит отрицательного числа равен 1. Для получения абсолютной величины числа нужно обнулить его

знаковый бит, не затрагивая остальные, тогда, независимо от знака исходного числа, результат всегда будет положительным. Это можно сделать при помощи операции логического И, в которой первым операндом выступает исходное число, а вторым — двоичная маска, имеющая в старшем разряде 0, а в остальных — 1.

**Листинг 14.14.** Вычисление абсолютных значений чисел с плавающей точкой двойной точности

```
.686
.model flat
option casemap: none
.XMM
.data
  msk       label qword
   msk_high  DQ 7FFFFFFFFFFFFFFFh
   msk_low   DQ 7FFFFFFFFFFFFFFFh
  res DQ 0
.code
 abs_ex proc
   push   EBP
   mov    EBP, ESP
   mov    ESI, dword ptr [EBP+8]
   lea    EDI, msk
   lea    EBX, res
   movupd XMM0, [ESI]
   movupd XMM1, [EDI]
   andpd  XMM0, XMM1
   movups [EBX], XMM0
   lea    EAX, res
   pop    EBP
   ret
 abs_ex endp
 end
```

Для 128-разрядного операнда, состоящего из двух упакованных 64-разрядных чисел с плавающей точкой, биты 127 и 63 содержат знак числа, поэтому в процедуре задана область памяти с меткой msk, представляющая собой 128-разрядное число с нулями в 127-м и 63-м битах. Для простоты область памяти msk представлена как два 64-разрядных числа — msk_high и msk_low.

В этой процедуре 128-разрядный операнд, адресуемый регистром ESI, помещается в регистр XMM0 с помощью команды

```
movupd XMM0, [ESI]
```

Маска msk помещается в регистр XMM1 при помощи команды

```
movupd XMM1, [EDI]
```

После этого абсолютное значение двух упакованных 64-разрядных операндов вычисляет команда

```
andpd XMM0, XMM1
```

Далее результат помещается в переменную res, и процедура возвращает ее адрес в регистре EAX.

В следующем примере я продемонстрирую, как можно поменять знак упакованного 64-разрядного операнда на противоположный. Для этого можно использовать процедуру sign_ex, принимающую в качестве входного параметра 128-разрядный операнд, а возвращающую в регистре EAX адрес памяти, где содержатся числа со знаком, противоположным знаку исходного операнда (листинг 14.15).

**Листинг 14.15.** Изменение знака упакованного 64-разрядного числа

```
.686
.model flat
option casemap: none
.XMM
.data
  msk        label qword
   msk_high  DQ 8000000000000000h
   msk_low   DQ 8000000000000000h
  res DQ 0
.code
 sign_ex proc
   push   EBP
   mov    EBP, ESP
   mov    ESI, dword ptr [EBP+8]
   lea    EDI, msk
   lea    EBX, res
   movupd XMM0, [ESI]
   movupd XMM1, [EDI]
   xorpd  XMM0, XMM1
   movups [EBX], XMM0
   lea    EAX, res
   pop    EBP
   ret
 sign_ex endp
 end
```

Проанализируем программный код процедуры. Из предыдущего примера понятно, как определяется знак числа. Для того чтобы изменить знак числа на противоположный, нужно выполнить операцию исключающего ИЛИ, в которой первым операндом выступает исходное число, а вторым — двоичная маска, имеющая в старшем разряде 1, а в остальных разрядах — 0.

Область памяти msk процедуры sign_ex содержит 128-разрядное значение, содержащее единицы в 127-м и 63-м разрядах. Разделение 128-разрядного значения на две переменные, msk_high и msk_low, делает принцип работы процедуры более понятным.

128-разрядный операнд, адресуемый регистром ESI, помещается в регистр XMM0 с помощью команды

```
movupd XMM0, [ESI]
```

Маска msk помещается в регистр XMM1 при помощи команды

```
movupd XMM1, [EDI]
```

После этого знаки двух упакованных 64-разрядных операндов изменяет команда

```
xorpd XMM0, XMM1
```

Далее результат помещается в переменную res, и процедура возвращает ее адрес в регистре EAX.

В следующую группу команд входят команды управления состоянием вычислений (ldmxcsr, stmxcsr, fxsave, fxrstor), которые были рассмотрены нами ранее в главе 13 при анализе SSE-расширения, а также команды управления кэшированием (clflush, lfence, mfence).

Назначение команд кэширования данных состоит в том, чтобы оптимизировать использование кэша данных при интенсивных вычислениях. Команда clflush обеспечивает установку признака обратной записи (write-back) неупорядоченных данных, что повышает производительность операций.

Команды lfence и mfence выполняют следующие функции:

- записывают некэшированные данные как полные строки в кэш данных;
- считывают некэшированные данные как полные строки, так, как если бы они находились в кэше данных.

# 14.2. Команды обработки 128-разрядных целочисленных данных

Помимо повышения эффективности обработки 128-разрядных упакованных данных с плавающей точкой двойной точности, SSE2-расширения повышают эффективность выполнения целочисленных операций. Многие MMX-команды целочисленного расширения были модифицированы таким образом, чтобы оперировать 128-разрядными значениями. Команды 128-разрядного целочисленного расширения имеют те же мнемонические обозначения, что и MMX-команды, но при работе с данными большей чем 64 бита разрядности добавляется суффикс. Команды для обработки 128-разрядных целых чисел позволяют повысить производительность выполнения приложений, в которых они используются, поскольку обрабатывают параллельно в два раза больше данных, чем MMX-команды.

Мнемоники операций в командах 128-разрядного целочисленного расширения остались теми же, что и в случае MMX. Компилятор определяет тип команды (версия для 64 или 128 бит) по типу регистров, используемых в качестве операндов.

Объясню это на примере. Предположим, что команда MMX-расширения имеет вид

```
команда mm0, mm1
```

В этом случае компилятор обрабатывает команду как имеющую MMX-формат. Еще пример:

```
команда xmm0, xmm1
```

Такую команду компилятор транслирует как имеющую 128-разрядный целочисленный формат SSE2.

Дополнительные инструкции 128-разрядного целочисленного расширения имеют в мнемонических обозначениях следующие символы:

- префикс p означает, что выполняется параллельная операция над элементами упакованных данных;
- суффиксы b, w, d, q, dq указывают на тип данных (байт, слово, двойное слово, учетверенное слово или двойное учетверенное слово);
- суффиксы s или u говорят о том, что операция выполняется со знаковыми (signed) или беззнаковыми (unsigned) значениями.

Например, обозначение команды pmuludq указывает на то, что выполняется операция умножения упакованных беззнаковых 16-байтовых (double-quadword) операндов. Рассмотрим более подробно команды 128-разрядного целочисленного расширения и начнем с команд пересылки данных.

В группу команд пересылки данных входят следующие команды:

- movq2dq — пересылка 64-разрядного операнда из MMX-регистра в XMM-регистр. Старшие 64 бита XMM-регистра заполняются нулями;
- movdq2q — пересылка младшей части 128-разрядного операнда из XMM-регистра в MMX-регистр;
- movdqa — пересылка 128-разрядных операндов. В качестве входного операнда (источника) могут выступать XMM-регистр или 128-разрядная ячейка памяти. Выходным операндом должен быть XMM-регистр. Данные должны быть выровнены по 16-байтовой границе, в противном случае возникает исключение общей защиты;
- movdqu — пересылка 128-разрядных операндов. В качестве входного операнда (источника) могут выступать XMM-регистр или 128-разрядная ячейка памяти. Выходным операндом должен быть XMM-регистр. Выравнивание данных по 16-байтовой границе необязательно.

К группе арифметических относятся следующие команды:

- paddb — сложение упакованных байтов операнда-источника и операнда-приемника. Результат помещается в операнд-приемник, в качестве которого выступает XMM-регистр. В качестве операнда-источника могут выступать либо XMM-регистр, либо ячейка памяти;
- paddw — сложение упакованных слов операнда-источника и операнда-приемника. Результат помещается в операнд-приемник, в качестве которого выступает XMM-регистр. В качестве операнда-источника могут выступать либо XMM-регистр, либо ячейка памяти;
- paddd — сложение упакованных двойных слов операнда-источника и операнда-приемника. Результат помещается в операнд-приемник, в качестве которого выступает XMM-регистр. В качестве операнда-источника могут выступать либо XMM-регистр, либо ячейка памяти;
- paddq — сложение упакованных учетверенных слов операнда-источника и операнда-приемника. Команда имеет две модификации: одна работает

с MMX-регистрами, другая — с XMM-регистрами. Результат помещается в операнд-приемник, в качестве которого выступает либо MMX-регистр, либо один из XMM-регистров. В качестве операнда-источника могут выступать MMX-регистр, XMM-регистр или ячейка памяти;

- psubb — вычитание упакованных байтов операнда-источника из операнда-приемника. Результат помещается в операнд-приемник, в качестве которого выступает XMM-регистр. В качестве операнда-источника могут выступать либо XMM-регистр, либо ячейка памяти;
- psubw — вычитание упакованных слов операнда-источника из операнда-приемника. Результат помещается в операнд-приемник, в качестве которого выступает XMM-регистр. В качестве операнда-источника могут выступать либо XMM-регистр, либо ячейка памяти;
- psubd — вычитание упакованных двойных слов операнда-источника из операнда-приемника. Результат помещается в операнд-приемник, в качестве которого выступает XMM-регистр. В качестве операнда-источника могут выступать либо XMM-регистр, либо ячейка памяти;
- psubq — вычитание упакованных учетверенных слов операнда-источника из операнда-приемника. Команда имеет две модификации: одна работает с MMX-регистрами, другая — с XMM-регистрами. Результат помещается в операнд-приемник, в качестве которого выступает либо MMX-регистр, либо один из XMM-регистров. В качестве операнда-источника могут выступать MMX- регистр, XMM-регистр или ячейка памяти;
- pmulhw — параллельное умножение упакованных слов операнда-источника и операнда-приемника. Старшие части произведений помещаются в операнд-приемник, в качестве которого может выступать один из XMM-регистров. Входным операндом, или источником, могут служить XMM-регистр или 128-разрядная ячейка памяти;
- pmullw — умножение упакованных слов операнда-источника и операнда-приемника. Младшие части произведений помещаются в операнд-приемник, в качестве которого может выступать один из XMM-регистров. Входным операндом, или источником, могут служить XMM-регистр или 128-разрядная ячейка памяти;
- pmuludq — умножение младших 32-разрядных целых чисел из 64-разрядных операндов источника и приемника. Результат умножения является 64-разрядной переменной целого типа. В качестве входного операнда могут выступать XMM-регистр или ячейка памяти, в качестве выходного — только XMM-регистр.

Рассмотрим пример применения арифметических операций для обработки 128-разрядных целочисленных данных. Здесь будут продемонстрированы выполнение различных команд 128-разрядного SSE2-расширения и техника совместного использования MMX- и SSE2-регистров.

Предположим, есть два массива 32-разрядных целых чисел (назовем их $a_1$ и $b_1$) и требуется вычислить выражение $(a_1 - b_1) \times (a_1 + b_1)$ для каждой пары элементов

массивов. Для вычисления значения этого выражения воспользуемся командами paddd, psubd и pmuldq, а также продемонстрируем работу команд пересылки. Программный код, выполняющий вычисления, реализован в виде процедуры int128_demo (листинг 14.16). Процедура принимает в качестве операндов адреса целочисленных массивов $a_1$ и $b_1$ и размер массивов в байтах, а в регистре EAX возвращает адрес массива, где хранится результат. Мнемоническое обозначение процедуры выглядит так:

```
int128_demo(address_a1. address_b1. size)
```

**Листинг 14.16.** Применение целочисленных 128-разрядных команд

```
.686
.model flat
.XMM
option casemap: none
.data
 res DQ 3 DUP(0)
.code
 int128_demo proc
   push     EBP
   mov      EBP. ESP
   mov      ESI. dword ptr [EBP+8]
   mov      EDI. dword ptr [EBP+12]
   lea      EBX. res
   mov      ECX. dword ptr [EBP+16]
   shr      ECX. 2
next:
   movd     MM0. dword ptr [ESI]   ; a1 -> MM0 (low 32 bit)
   movq2dq  XMM0. MM0              ; a1 -> XMM0 (low 32 bit)
   movdqu   XMM2. XMM0             ; save XMM0 in XMM2
   movd     MM0. dword ptr [EDI]   ; b1 -> MM0
   movq2dq  XMM1. MM0              ; b1 -> XMM0 (low 32 bit)
   psubd    XMM0. XMM1             ; a1-b1 -> XMM0
   paddd    XMM2. XMM1             ; a1+b1 -> XMM2
   pmuludq  XMM0. XMM2             ; (a1-b1)×(a1+b1) -> XMM0
   movdq2q  MM0. XMM0
   movq     [EBX]. MM0
   add      ESI. 4
   add      EDI. 4
   add      EBX. 8
   dec      ECX
   jnz      next
   pop      EBP
   lea      EAX. res
   ret
 int128_demo endp
 end
```

Работа процедуры начинается с загрузки адресов массивов при помощи команд

```
mov      ESI. dword ptr [EBP+8]
mov      EDI. dword ptr [EBP+12]
lea      EBX. res
```

Для большей определенности положим, что регистр ESI содержит адрес массива $a_1$, регистр EDI — адрес массива $b_1$ и EBX — адрес массива res, где будет сохранен результат. Массив res в нашем случае содержит четыре 32-разрядных элемента суммарным размером в учетверенное слово (8 байт), хотя можно выбрать и другой размер. Элементы массива res представляют собой произведение 32-разрядных операндов, и для каждого из них резервируется 64-разрядная ячейка памяти. Следующие команды помещают в регистр ECX количество двойных слов массивов $a_1$ и $b_1$ (размеры массивов предполагаются одинаковыми):

```
mov  ECX, dword ptr [EBP+16]
shr  ECX, 2
```

Затем в цикле next осуществляется вычисление выражения для каждой пары элементов массивов $a_1$ и $b_1$. В каждой итерации цикла выполняются следующие действия:

1. В младшее двойное слово MMX-регистра MM0 помещается 32-разрядное целое число из массива $a_1$. Для этого служит команда

   ```
   movd MM0, dword ptr [ESI]
   ```

2. Чтобы можно было использовать команды 128-разрядной арифметики, 32-разрядный операнд из регистра MM0 помещается в регистр XMM0 командой

   ```
   movq2dq XMM0, MM0
   ```

   При этом старшие разряды регистра XMM0 заполняются нулями.

3. Содержимое регистра XMM0 для последующего использования копируется в регистр XMM2 командой

   ```
   movdqu XMM2, XMM0
   ```

4. В младшее двойное слово MMX-регистра MM0 помещается 32-разрядное целое число из массива $b_1$ командой

   ```
   movd MM0, dword ptr [EDI]
   ```

5. 32-разрядный операнд из регистра MM0 помещается в регистр XMM0 командой

   ```
   movq2dq XMM1, MM0
   ```

6. Вычисляется разность $a_1 - b_1$. Для этого используется команда

   ```
   psubd XMM0, XMM1
   ```

   Результат сохраняется в регистре XMM0.

7. Вычисляется сумма $a_1 + b_1$. Для этого используется команда

   ```
   paddd XMM2, XMM1
   ```

   Результат сохраняется в регистре XMM2.

8. Значение выражения $(a_1 - b_1) \times (a_1 + b_1)$ вычисляется при помощи команды

   ```
   pmuludq XMM0, XMM2
   ```

   При этом результат сохраняется в регистре XMM0.

9. Полученный результат сохраняется в соответствующем элементе массива res с помощью команд

```
movdq2q MM0. XMM0
movq   [EBX]. MM0
```

По завершении всех итераций адрес массива res помещается в регистр EAX, после чего происходит выход из процедуры.

Работоспособность процедуры легко проверить при помощи программы на Visual C++ .NET (листинг 14.17).

**Листинг 14.17.** Демонстрационная программа для процедуры из листинга 14.16

```
#include <stdio.h>
extern "C" long long* int128_demo(int* a1. int* a2. int asize);
int main(void)
{
 int a1[3] = { 1259. 954. -6451 };
 int a2[3] = { -2901. -419. 4202 };
 int asize = sizeof(a1);
 long long* pi1 = int128_demo(a1. a2. asize);
 printf("INT-128 DEMO results:\n");
 for (int i1 = 0; i1 < asize / 4; i1++)
  {
   printf("%d ". *pi1++);
  }
 return 0;
}
```

В этой программе процедура int128_demo должна быть объявлена с директивой extern. Обратите внимание на то, что процедура оперирует целочисленными значениями (тип int), имеющими разрядность 32 бита, в то время как произведения $(a_1 - b_1) \times (a_1 + b_1)$ имеют 64-разрядную разрядность. По этой причине выходное значение (адрес массива), возвращаемое процедурой, объявлено как имеющее тип long long*.

SSE2-расширение включает новые команды сдвига целочисленных операндов:

- pslldq — сдвиг влево содержимого XMM-регистра (операнд-приемник) на указанное вторым операндом количество байтов. Второй операнд является непосредственным значением. Обратите внимание на то, что сдвиг осуществляется не на биты, а на байты. Схема работы команды pslldq показана на рис. 14.8;

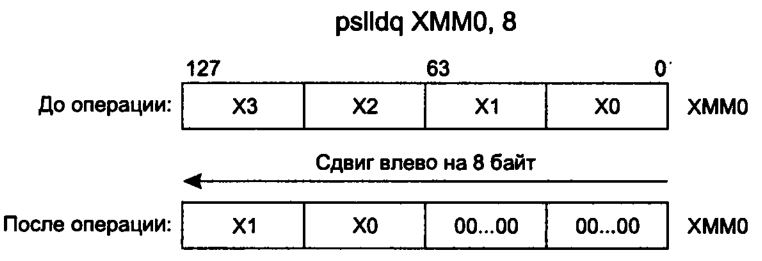



**Рис. 14.8.** Схема работы команды pslldq XMM0, 8

* psrldq — сдвиг влево содержимого XMM-регистра (операнд-приемник) на указанное вторым операндом количество байтов. Второй операнд является непосредственным значением. Обратите внимание на то, что сдвиг осуществляется не на биты, а на байты. Схема работы команды psrldq показана на рис. 14.9.

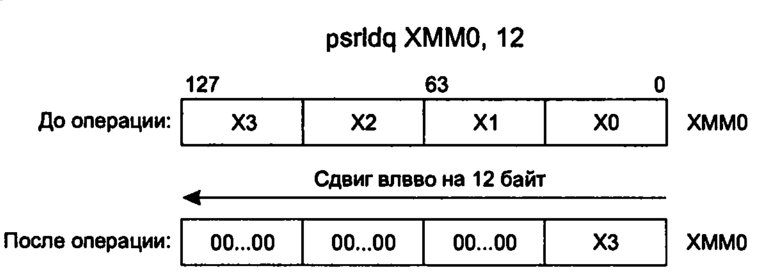



**Рис. 14.9.** Схема работы команды psrldq XMM0, 12

SSE2-расширение включает новые команды перестановки целочисленных операндов:

* pshufd — перестановка и сохранение четырех 32-разрядных элементов операнда-источника, представляющего собой XMM-регистр или ячейку памяти, в операнд-приемник, в качестве которого выступает XMM-регистр. Порядок перестановки определяется 8-разрядной маской, являющейся третьим операндом. Содержимое операнда-источника при этом не изменяется. Схема работы команды pshufd показана на рис. 14.10;

pshufd XMM0, XMM1, 9Ch

До операции:

| 127 | | 63 | 0 | |
|---|---|---|---|---|
| X3 | X2 | X1 | X0 | XMM0 |

| 127 | | 63 | 0 | |
|---|---|---|---|---|
| Y3 | Y2 | Y1 | Y0 | XMM1 |

После операции:

| 127 | | 63 | 0 | |
|---|---|---|---|---|
| Y2 | Y1 | Y3 | Y0 | XMM0 |

Маска: 9Ch = | 10 | 01 | 11 | 00 |

**Рис. 14.10.** Схема работы команды pshufd XMM0, XMM1, 9Ch

* pshuflw — перестановка и сохранение младших слов операнда-источника, представляющего собой XMM-регистр или ячейку памяти, в операнд-приемник, в качестве которого выступает XMM-регистр. Порядок перестановки

определяется 8-разрядной маской, являющейся третьим операндом. Содержимое операнда-источника при этом не изменяется. Схема работы команды pshuflw показана на рис. 14.11;

pshuflw XMM0, XMM1, 9Ch

До операции:

| 127 | 63 | 47 | 31 | 15 0 | |
|---|---|---|---|---|---|
| X3 | X3 | X2 | X1 | X0 | XMM0 |
| Y3 | Y3 | Y2 | Y1 | Y0 | XMM1 |

После операции:

| 127 | 63 | 47 | 31 | 15 0 | |
|---|---|---|---|---|---|
| X3 | Y2 | Y1 | Y3 | Y0 | XMM0 |

Маска: 9Ch = | 10 | 01 | 11 | 00 |

**Рис. 14.11.** Схема работы команды pshuflw XMM0, XMM1, 9Ch

- pshufhw — перестановка и сохранение старших слов операнда-источника, представляющего собой XMM-регистр или ячейку памяти, в операнд-приемник, в качестве которого выступает XMM-регистр. Порядок перестановки определяется 8-разрядной маской, являющейся третьим операндом. Содержимое операнда-источника при этом не изменяется. Схема работы команды pshufhw показана на рис. 14.12.

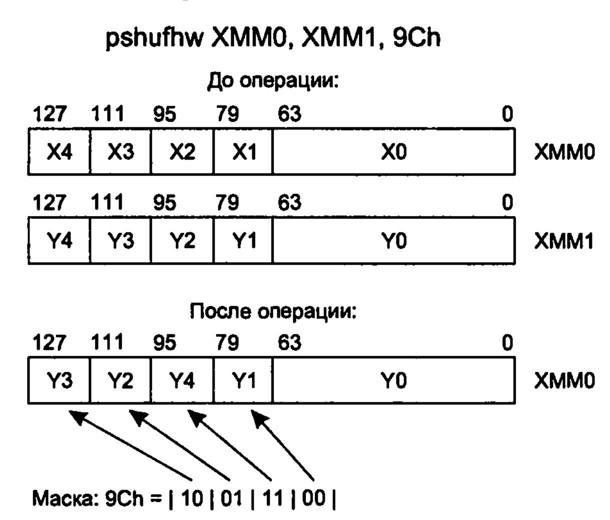



**Рис. 14.12.** Схема работы команды pshufhw XMM0, XMM1, 9Ch

Рассмотрим использование команд перестановки на следующем примере. Предположим, имеется массив целых чисел $a_1$, содержащий элементы $x_1$, $x_2$, $x_3$, ..., $xn$. Требуется изменить порядок следования операндов в массиве на обратный, то есть первым элементом массива должен быть $xn$, вторым — $xn_{-1}$ и, наконец, последним элементом — $x_1$. Программный код, реализующий этот алгоритм, представлен в виде демонстрационной процедуры (назовем ее _pshuf_lh_ex) в листинге 14.18.

**Листинг 14.18.** Изменение порядка следования элементов массива

```
.686
.model flat
.XMM
option casemap:none
.data
  res  DW 8 DUP (0)
.code
_pshuf_lh_ex proc
  push    EBP
  mov     EBP, ESP
  mov     ESI, dword ptr [EBP+8]
  lea     EDI, res
  movdqu  XMM0, [ESI]
  pslldq  XMM0, 8
  movdqu  XMM1, [ESI]
  psrldq  XMM1, 8
  paddw   XMM1, XMM0
  pshuflw XMM1, XMM1, 1Bh
  pshufhw XMM1, XMM1, 1Bh
  movdqu  [EDI], XMM1
  pop     EBP
  lea     EAX, res
  ret
_pshuf_lh_ex endp
end
```

Проанализируем программный код процедуры. В качестве входного параметра процедура принимает адрес массива 16-разрядных чисел, который помещается в регистр ESI. В регистре EAX процедура возвращает адрес массива res, в котором будет храниться результат перестановки элементов исходного массива.

Предположим для упрощения, что исходный массив (массив $a_1$) содержит целочисленные значения $x_0$–$x_7$, показанные на рис. 14.13.

**Массив a1 из 8 элементов**

| x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

**Рис. 14.13.** Содержимое элементов массива $a_1$

128-разрядный элемент массива $a_1$ в регистр XMM0 помещает команда

```
movdqu XMM0, [ESI]
```

Замечу, что эта команда не требует выравнивания адреса операнда по 16-байтовой границе. Следующая за ней команда выполняет сдвиг содержимого регистра XMM0 на 8 байт влево:

```
pslldq XMM0, 8
```

После выполнения этой операции регистр XMM0 будет содержать значение, показанное на рис. 14.14.

| 127 | 111 | 95 | 79 | 63 | | | 0 | |
|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | XMM0 |

**Рис. 14.14.** Содержимое регистра XMM0 после выполнения команды pslldq

Далее воспользумся регистром XMM1. После того как будут выполнены следующие команды, регистр XMM1 будет содержать значение, показанное на рис. 14.15:

```
movdqu  XMM1, [ESI]
psrldq  XMM1, 8
```

| 127 | | | | 63 | 47 | 31 | 15 0 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 7 | 6 | 5 | 4 | XMM1 |

**Рис. 14.15.** Содержимое регистра XMM1 после выполнения команды psrldq

Затем выполняем операцию сложения упакованных целых чисел командой

```
paddw XMM1, XMM0
```

Содержимое регистра-приемника XMM1 после этой операции будет таким, как показано на рис. 14.16.

| 127 | | | | 63 | 47 | 31 | 15 0 | |
|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | XMM1 |

**Рис. 14.16.** Содержимое регистра XMM1 после выполнения команды paddw

Как видно из рис. 14.16, старшие 4 слова регистра XMM1 содержат младшие слова массива $a_1$, а младшие 4 слова регистра — старшие слова $a_1$. Следующие две команды располагают элементы регистра XMM1 в нужном порядке:

```
pshuflw XMM1, XMM1, 1Bh
pshufhw XMM1, XMM1, 1Bh
```

Рисунок 14.17 иллюстрирует работу одной из этих команд:

```
pshuflw XMM1, XMM1, 1Bh
```

После выполнения этих двух команд в регистре XMM1 будет находиться последовательность чисел, представленная рис. 14.18.

К этому моменту регистр XMM1 содержит элементы массива $a_1$, расположенные в обратном порядке. Теперь остается сохранить результат в массиве res (команда movdqu [EDI], XMM1) и передать адрес этого массива в вызывающую программу (команда lea EAX, res).




**Рис. 14.17.** Содержимое регистра XMM1 после выполнения команды pshuflw

127 ... 0

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | XMM1 |
|---|---|---|---|---|---|---|---|---|

**Рис. 14.18.** Содержимое регистра XMM1 после выполнения всех преобразований

Работоспособность процедуры легко проверить при помощи программы на Visual C++ .NET (листинг 14.19).

**Листинг 14.19.** Демонстрационная программа для процедуры из листинга 14.18

```
#include <stdio.h>
extern "C" short int* pshuf_lh_ex(short int* a1);
int main(void)
{
  __declspec(align(16)) short int a1[8] = { 0, 1, 2, 3, 4, 5, 6, 7 };
  short int* pa1 = pshuf_lh_ex(a1);
  printf("PSHUFHW PSHUFLW example:\n");
  for (int i1 = 0; i1 < 8; i1++)
   {
    printf("%d ", *pa1++);
   }
  return 0;
}
```

В этой программе a1 — массив 16-разрядных целых чисел со знаком (short int). Процедура pshuf_lh_ex объявлена с директивой extern и в качестве параметра принимает указатель на массив 16-разрядных целых чисел (short int*), а возвращает адрес массива, в котором хранятся результаты (short int*). Программа выводит на экран такой результат:

```
PSHUFHW PSHUFLW example:
7 6 5 4 3 2 1 0
```

К группе команд распаковки относятся команды, которые мнемонически можно обозначить как punpckh и punpckl. С помощью этих команд проводится распаковка и перестановка старших (младших) частей операнда-источника и операнда-приемника. При этом младшие (старшие) части операндов игнорируются.

В качестве операндов-источников могут выступать XMM-регистры или 128-разрядные ячейки памяти, в качестве операндов-приемников — только XMM-регистры. К этим командам относятся:

- punpckhbw — распаковка старших байтов операнда-источника и операнда-приемника в старшие и младшие байты слов операнда-приемника соответственно;
- punpcklbw — распаковка младших байтов операнда-источника и операнда-приемника в старшие и младшие байты слов операнда-приемника соответственно;
- punpckhwd — распаковка старших слов операнда-источника и операнда-приемника в старшие и младшие слова двойных слов операнда-приемника соответственно;
- punpcklwd — распаковка младших слов операнда-источника и операнда-приемника в старшие и младшие слова двойных слов операнда-приемника соответственно;
- punpckhdq — распаковка старших двойных слов операнда-источника и операнда-приемника в старшие и младшие двойные слова учетверенных слов операнда-приемника соответственно;
- punpckldq — распаковка младших двойных слов операнда-источника и операнда-приемника в старшие и младшие двойные слова учетверенных слов операнда-приемника соответственно;
- punpckhqdq — распаковка старших учетверенных слов операнда-источника и операнда-приемника в старшие и младшие учетверенные слова двойных учетверенных слов операнда-приемника соответственно;
- punpcklqdq — распаковка младших учетверенных слов операнда-источника и операнда-приемника в старшие и младшие учетверенные слова двойных учетверенных слов операнда-приемника соответственно.

На этом рассмотрение SSE2-расширения можно закончить. Мы познакомились с большинством команд и способами их применения в программах на ассемблере, хотя эта тема очень обширна и ей посвящены многочисленные публикации. Основным источником информации по этим вопросам является документация фирмы Intel, в которой технология SSE2 описывается более подробно.

# Заключение

Прогресс в индустрии высокопроизводительных процессоров столь стремителен, что новые технологии появляются до того, как устаревают старые. Линейка процессоров Intel Pentium 4 совсем недавно пополнилась новыми устройствами — были выпущены семь новых процессоров, среди которых четыре представителя с новой микроархитектурой ядра (Prescott) и кэшем 2-го уровня размером в 1 Мбайт.

Все эти процессоры рассчитаны на шину с частотой 800 МГц и поддерживают гиперпотоковую (hyper-threading) технологию. Кроме того, фирма Intel выпустила Pentium 4 на ядре Prescott с частотой 2,8 ГГц, как и предыдущий изготовленный по 90-нанометровой технологии, но рассчитанный на частоту 533 МГц и не поддерживающий гиперпотоковую технологию. По информации Intel, предназначен этот процессор специально для производителей комплексного оборудования.

Но самым важным, с точки зрения программиста, было то, что в новом ядре Prescott появилось очередное расширение, получившее название SSE3 и содержащее 13 новых команд. Все они, за исключением трех, используют SSE-регистры и предназначены для повышения производительности при выполнении следующих операций:

- быстрое преобразование вещественного числа в целое (соответствующая команда fisttp заменяет семь «обычных» команд);
- сложные арифметические вычисления (команды addsubps, addsubpd, movsldup, movshdup, movddup);
- кодирование видео (команда lddqu);
- обработка графики (команды haddps, hsubps, haddpd, hsubpd);
- синхронизация потоков (команды monitor, mwait).

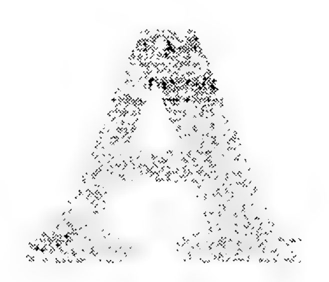
Детальное рассмотрение новых команд выходит за рамки материала книги, но можно дать их краткий обзор.

Инструкции первых четырех категорий, помимо выполнения самих операций, позволяют экономить ресурсы процессора, что, в свою очередь, оптимизирует работу программных потоков и механизма спекулятивного выполнения команд. Программный код при этом также значительно сокращается и упрощается. Даже по сравнению с быстрыми SSE2-командами SSE3-команды во многих случаях более экономичны.

Две команды последней группы — monitor и mwait — позволяют потоку работающего приложения сообщать процессору, что в данный момент он не выполняет полезной работы и находится в режиме ожидания. В этом случае процессор может перейти в режим пониженного энергопотребления или (если используется гиперпотоковая технология) отдать все ресурсы другому программному потоку.

Обобщая, можно смело утверждать, что с появлением технологии SSE3 перед программистами открываются новые возможности по оптимизации кода и разработке еще более производительных приложений.

# Базовые инструкции процессоров 80x86




Это приложение является справочником по базовой системе команд семейства процессоров Intel. В справочник включены команды для моделей процессоров 80386 и более поздних. Для описания форматов команд используется ряд аббревиатур, представленных в табл. А.1. Сами команды описаны в табл. А.2.

**Таблица А.1.** Аббревиатуры команд

| Обозначение | Краткое описание |
|---|---|
| Reg | Один из 8-, 16- или 32-разрядных регистров из списка: AH, AL, BH, BL, CH, CL, AX, BX, CX, DX, SI, DI, BP, SP, EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP |
| reg8, reg16, reg32 | Регистр общего назначения, определяемый количеством битов |
| Acc | AL, AX или EAX |
| Mem | Операнд в памяти |
| mem8, mem16, mem32 | Операнд в памяти, определяемый количеством битов |
| Immed | Непосредственный операнд |
| immed8, immed16, immed32 | Непосредственный операнд с определенным количеством битов |
| Label | Метка |

**Таблица А.2.** Система команд

| Код операции | Операнды | Функция |
|---|---|---|
| aaa | | ASCII-коррекция после сложения |
| aad | | ASCII-коррекция перед делением |
| aam | | ASCII-коррекция после умножения |

продолжение ⇗

**Таблица А.2** (продолжение)

| **Код операции** | **Операнды** | **Функция** |
|---|---|---|
| aas | | ASCII-коррекция после вычитания |
| adc | reg, reg | Сложение с переносом |
| | mem, reg | |
| | reg, mem | |
| | reg, immed | |
| | mem, immed | |
| | acc, immed | |
| add | reg, reg | Сложение |
| | mem, reg | |
| | reg, mem | |
| | reg, immed | |
| | mem, immed | |
| | acc, immed | |
| and | reg, reg | Логическое И |
| | mem, reg | |
| | reg, mem | |
| | reg, immed | |
| | mem, immed | |
| | acc, immed | |
| bsf, bsr | reg16, reg16 | Сканирование битов |
| | reg16, mem16 | |
| | reg32, reg32 | |
| | reg32, mem32 | |
| bt, btc, btr, bts | reg16, immed8 | Проверка битов |
| | reg16, reg16 | |
| | mem16, immed8 | |
| | mem16, reg16 | |
| call | label | Вызов процедуры |
| | reg | |
| | mem16 | |
| | mem32 | |
| cbw | | Преобразование байта в слово |
| cdq | | Преобразование двойного слова в учетверенное |
| clc | | Сброс флага переноса |

| Код операции | Операнды | Функция |
|---|---|---|
| cld | | Сброс флага направления |
| cli | | Сброс флага прерывания |
| cmc | | Инвертирование флага переноса |
| cmp | | Сравнение операндов |
| cmps, cmpsb, cmpsw, cmpsd | mem, mem | Сравнение строк |
| cwd | | Преобразование слова в двойное слово |
| daa | | Десятичная коррекция после сложения |
| das | | Десятичная коррекция после вычитания |
| dec | reg | Декремент |
| | mem | |
| div | reg | Деление без знака |
| | mem | |
| idiv | reg | Деление целых чисел со знаком |
| | mem | |
| imul | reg | Умножение целых чисел со знаком |
| | mem | |
| in | acc, immed | Ввод из порта |
| inc | reg | Инкремент |
| | mem | |
| int | | Генерирование программного прерывания |
| iret | | Возврат из прерывания |
| Jcondition | label | Переход, если выполнено условие |
| jmp | label | Безусловный переход |
| lahf | | Загрузка флагов в AH |
| lds, les, lfs, lgs, lss | | Загрузка дальнего указателя |
| lea | reg, mem | Загрузка текущего адреса |
| lods, lodsb, lodsw, lodsd | mem | Загрузка строки в аккумулятор |
| loop | label | Цикл, в котором выполняется декремент регистра CX и переход на метку, пока CX больше 0 |
| loope, loopz | label | Цикл, если равно 0. Декремент регистра CX и переход на метку, если CX больше 0 и флаг нуля установлен |
| loopne, loopz | label | Цикл, если не равно 0. Декремент регистра CX и переход на метку, если CX больше 0 и флаг нуля сброшен |

продолжение ⇗

**Таблица А.2** (продолжение)

| **Код операции** | **Операнды** | **Функция** |
|---|---|---|
| mov | reg, reg<br>mem, reg<br>reg, mem<br>reg, immed<br>mem, immed | Пересылка операндов |
| movs, movsb, movsw, movsd | mem, mem | Пересылка строк |
| mul | reg<br>mem | Умножение целых чисел без знака |
| neg | reg<br>mem | Изменение знака операнда |
| nop | | Отсутствие операций; используется для организации задержек в циклах |
| not | reg<br>mem | Логическое НЕ; инвертирование каждого бита операнда |
| or | reg, reg<br>mem, reg<br>reg, mem<br>reg, immed<br>mem, immed<br>acc, immed | Логическое ИЛИ |
| out | immed, acc<br>DX, acc | Вывод в порт |
| pop | reg16<br>reg32<br>mem16<br>mem32 | Извлечение операнда из стека |
| popa, popad | | Извлечение из стека регистров общего назначения (popa — 16-разрядных, popad — 32-разрядных) |
| popf, popfd | | Извлечение флагов из стека |
| push | reg16<br>reg32<br>mem16<br>mem32 | Помещение в стек |
| pusha, pushad | | Помещение в стек всех регистров |

| Код операции | Операнды | Функция |
|---|---|---|
| pushf, pushfd | | Помещение регистра флагов в стек |
| rcl | reg, immed8<br>reg, CL<br>mem, immed8<br>mem, CL | Циклический сдвиг операнда влево через флаг переноса |
| rcr | reg, immed8<br>reg, CL<br>mem, immed8<br>mem, CL | Циклический сдвиг операнда вправо через флаг переноса |
| rep | | Повторение команды для строкового примитива с использованием регистра CX как счетчика |
| repcondition | | Повторение команд строковых примитивов по условию |
| ret | | Возврат из процедуры |
| retn | immed8 | Возврат из процедуры с восстановлением стека. Непосредственный операнд определяет значение, которое должно быть добавлено к регистру-указателю стека |
| rol | reg, immed8<br>reg, CL<br>mem, immed8<br>mem, CL | Циклический сдвиг влево |
| ror | reg, immed8<br>reg, CL<br>mem, immed8<br>mem, CL | Циклический сдвиг вправо |
| sahf | | Загрузка регистра флагов из регистра AH |
| sal | reg, immed8<br>reg, CL<br>mem, immed8<br>mem, CL | Арифметический сдвиг влево |
| sar | reg, immed8<br>reg, CL<br>mem, immed8<br>mem, CL | Арифметический сдвиг вправо |
| sbb | reg, reg<br>mem, reg<br>reg, mem<br>reg, immed<br>mem, immed | Вычитание с заемом |

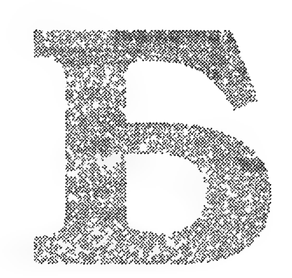
продолжение ⇨

**Таблица А.2** (продолжение)

| Код операции | Операнды | Функция |
|---|---|---|
| scas, scasb, scasw, scasd | mem | Сканирование строки со сравнением значений элементов со значением в аккумуляторе |
| SETcondition | reg8 | Установка по условию. Если заданное условие истинно, то байт-получатель устанавливается в 1, если ложно — в 0 |
| | mem8 | |
| shl | reg, immed8 | Логический сдвиг влево |
| | reg, CL | |
| | mem, immed8 | |
| | mem, CL | |
| shr | reg, immed8 | Логический сдвиг вправо |
| | reg, CL | |
| | mem, immed8 | |
| | mem, CL | |
| stc | | Установка флага переноса |
| std | | Установка флага направления |
| sti | | Установка флага прерывания |
| stos, stosb, stosw, stosd | mem | Сохранение содержимого аккумулятора в ячейке памяти, принадлежащей буферу строки |
| sub | reg, reg | Вычитание |
| | mem, reg | |
| | reg, mem | |
| | reg, immed | |
| | mem, immed | |
| | acc, immed | |
| test | reg, reg | Проверка отдельных битов операнда-получателя с соответствующими битами операнда-приемника. Выполняется операция логического И, в результате устанавливаются соответствующие флаги |
| | mem, reg | |
| | reg, mem | |
| | reg, immed | |
| | mem, immed | |
| | acc, immed | |
| wait | | Приостановка процессора |
| xchg | reg, reg | Обмен содержимым операнда-отправителя и операнда-получателя |
| | mem, reg | |
| | reg, mem | |

| Код операции | Операнды | Функция |
|---|---|---|
| xlat, xlatb | mem | Использование значения в регистре AL как индекса таблицы, на которую указывает содержимое регистра BX |
| xor | reg, reg | Логическое исключающее ИЛИ |
| | mem, reg | |
| | reg, mem | |
| | reg, immed | |
| | mem, immed | |
| | acc, immed | |

# Специальные инструкции процессоров 80x86



Это приложение является справочником по специальным командам семейства процессоров Intel Pentium. Для описания форматов команд используется ряд аббревиатур, представленных в табл. Б.1. Сами команды описаны в табл. Б.2 и Б.3.

**Таблица Б.1.** Аббревиатуры команд

| Обозначение | Краткое описание |
|---|---|
| Reg | Один из 8-, 16- или 32-разрядных регистров из списка: AH, AL, BH, BL, CH, CL, AX, BX, CX, DX, SI, DI, BP, SP, EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP |
| reg8, reg16, reg32 | Регистр общего назначения, определяемый количеством битов |
| Acc | AL, AX или EAX |
| Mem | Операнд в памяти |
| mem8, mem16, mem32 | Операнд в памяти, определяемый количеством битов |
| Immed | Непосредственный операнд |
| immed8, immed16, immed32 | Непосредственный операнд с определенным количеством битов |
| Label | Метка |

**Таблица Б.2.** Команды setCC процессоров Intel Pentium

| Код операции | Операнды | Функция |
|---|---|---|
| SETAE/SETNB | mem8/reg8 | Установить, если больше или равно/не меньше |
| SETE/SETZ | mem8/reg8 | Установить, если равно/нуль |
| SETNE/SETNZ | mem8/reg8 | Установить, если не равно/не нуль |
| SETB/SETNAE | mem8/reg8 | Установить, если меньше/не больше или равно |
| SETBE/SETNA | mem8/reg8 | Установить, если меньше или равно/не больше |
| SETL/SETNGE | mem8/reg8 | Установить, если меньше/не больше или равно |
| SETGE/SETNL | mem8/reg8 | Установить, если больше или равно/не меньше |

| Код операции | Операнды | Функция |
|---|---|---|
| SETG/SETNLE | mem8/reg8 | Установить, если больше/не меньше или равно |
| SETS | mem8/reg8 | Установить, если флаг SF = 1 |
| SETNS | mem8/reg8 | Установить, если флаг SF = 0 |
| SETC | mem8/reg8 | Установить, если флаг CF = 1 |
| SETNC | mem8/reg8 | Установить, если флаг CF = 0 |
| SETO | mem8/reg8 | Установить, если флаг OF = 1 |
| SETNO | mem8/reg8 | Установить, если флаг OF = 0 |
| SETP/SETPE | mem8/reg8 | Установить, если флаг PF = 1 |
| SETNP/SETPO | mem8/reg8 | Установить, если флаг PF = 0 |

**Таблица Б.3.** Команды cmovCC процессоров Intel Pentium

| Код операции | Операнды | Функция |
|---|---|---|
| CMOVA/CMOVNBE | reg16/32, mem16/32<br>reg16/32, reg16/32 | Переслать, если больше/не меньше или равно |
| CMOVAE/CMOVNB | reg16/32, mem16/32<br>reg16/32, reg16/32 | Переслать, если больше или равно/не меньше |
| CMOVNC | reg16/32, mem16/32<br>reg16/32, reg16/32 | Переслать, если флаг переноса CF не установлен |
| CMOVB/CMOVNAE | reg16/32, mem16/32<br>reg16/32, reg16/32 | Переслать, если меньше/не больше или равно |
| CMOVC | reg16/32, mem16/32<br>reg16/32, reg16/32 | Переслать, если флаг переноса CF установлен |
| CMOVBE/CMOVNA | reg16/32, mem16/32<br>reg16/32, reg16/32 | Переслать, если меньше или равно/не больше |
| CMOVE/CMOVZ | reg16/32, mem16/32<br>reg16/32, reg16/32 | Переслать, если равно или установлен флаг ZF |
| CMOVNE/CMOVNZ | reg16/32, mem16/32<br>reg16/32, reg16/32 | Переслать, если не равно или не установлен флаг ZF |
| CMOVP/CMOVPE | reg16/32, mem16/32<br>reg16/32, reg16/32 | Переслать, если установлен флаг PF |
| CMOVNP/CMOVPO | reg16/32, mem16/32<br>reg16/32, reg16/32 | Переслать, если не установлен флаг PF |
| CMOVGE/CMOVNL | reg16/32, mem16/32<br>reg16/32, reg16/32 | Переслать, если больше или равно/не меньше |
| CMOVL/CMOVNGE | reg16/32, mem16/32<br>reg16/32, reg16/32 | Переслать, если меньше/не больше или равно |
| CMOVLE/CMOVNG | reg16/32, mem16/32<br>reg16/32, reg16/32 | Переслать, если меньше или равно/не больше |

продолжение ⇗

**Таблица Б.3** (продолжение)

| **Код операции** | **Операнды** | **Функция** |
|---|---|---|
| CMOVO | reg16/32, mem16/32 | Переслать, если установлен флаг OF |
| | reg16/32, reg16/32 | |
| CMOVNO | reg16/32, mem16/32 | Переслать, если не установлен флаг OF |
| | reg16/32, reg16/32 | |
| CMOVS | reg16/32, mem16/32 | Переслать, если установлен флаг SF |
| | reg16/32, reg16/32 | |
| CMOVNS | reg16/32, mem16/32 | Переслать, если не установлен флаг SF |
| | reg16/32, reg16/32 | |

# Список литературы

1. Desktop Performance and Optimization for Pentium® 4 Processor, Intel® Corp., 2001.
2. IA-32 Intel Architecture Software Developer's Manual, Intel® Corp., 2001.
3. IA-32 Intel Architecture Optimization, Intel® Corp., 2001.
4. Ирвин К. Язык ассемблера для процессоров Intel. 3-е изд. / Пер. с англ. — М.: Издательский дом «Вильямс», 2002.
5. Магда Ю. С. Ассемблер. Разработка и оптимизация Windows-приложений. — БХВ-Петербург, 2003.
6. Юров В. И. Ассемблер. — СПб.: Питер, 2002.

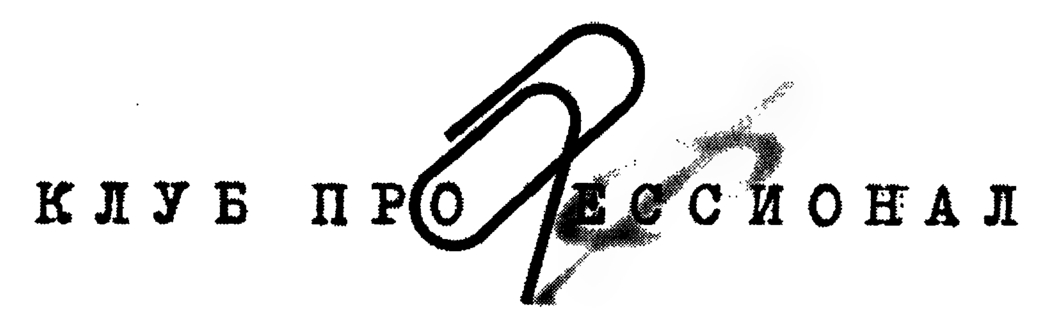
*Юрий Степанович Магда*

**Ассемблер для процессоров Intel Pentium**

| | |
|---|---|
| Заведующий редакцией | *А. Кривцов* |
| Ведущий редактор | *А. Адаменко* |
| Литературный редактор | *А. Жданов* |
| Художник | *Л. Адуевская* |
| Иллюстрации | *Г. Домрачева, Л. Родионова* |
| Корректоры | *И. Тимофеева, Н. Филатова* |
| Верстка | *Р. Гришанов* |

Подписано в печать 21.02.06. Формат 70×100/16. Усл. п. л. 33,54. Тираж 3000 экз. Заказ № 504.
ООО «Питер Пресс», Санкт-Петербург, Петергофское шоссе, д. 73, лит. А29.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 953005 — литература учебная

Отпечатано с готовых диапозитивов в ФГУП «Печатный двор» им. А. М. Горького
Федерального агентства по печати и массовым коммуникациям.
197110, Санкт-Петербург, Чкаловский пр., 15.


КЛУБ ПРОФЕССИОНАЛ




**УВАЖАЕМЫЕ ГОСПОДА!**
**КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»**
**ВЫ МОЖЕТЕ ПРИОБРЕСТИ**
**ОПТОМ И В РОЗНИЦУ**
**У НАШИХ РЕГИОНАЛЬНЫХ ПАРТНЕРОВ.**

**Башкортостан**
Уфа, «Азия», ул. Гоголя, д. 36, офис 5,
тел./факс (3472) 50-39-00, 51-85-44.
E-mail: asiaufa@ufanet.ru

**Дальний Восток**
Владивосток, «Приморский торговый дом книги»,
тел./факс (4232) 23-82-12.
E-mail: bookbase@mail.primorye.ru

Хабаровск, «Мирс»,
тел. (4212) 30-54-47, факс 22-73-30.
E-mail: sale_book@bookmirs.khv.ru

Хабаровск, «Книжный мир»,
тел. (4212) 32-85-51, факс 32-82-50.
E-mail: postmaster@worldbooks.kht.ru

**Европейские регионы России**
Архангельск, «Дом книги»,
тел. (8182) 65-41-34, факс 65-41-34.
E-mail: book@atnet.ru

Калининград, «Вестер»,
тел./факс (0112) 21-56-28, 21-62-07.
E-mail: nshibkova@vester.ru
http://www.vester.ru

**Северный Кавказ**
Ессентуки, «Россы», ул. Октябрьская, 424,
тел./факс (87934) 6-93-09.
E-mail: rossy@kmw.ru

**Сибирь**
Иркутск, «ПродаЛитЪ»,
тел. (3952) 59-13-70, факс 51-30-70.
E-mail: prodalit@irk.ru
http://www.prodalit.irk.ru

Иркутск, «Антей-книга»,
тел./факс (3952) 33-42-47.
E-mail: antey@irk.ru

Красноярск, «Книжный мир»,
тел./факс (3912) 27-39-71.
E-mail: book-world@public.krasnet.ru

Нижневартовск, «Дом книги»,
тел. (3466) 23-27-14, факс 23-59-50.
E-mail: book@nvartovsk.wsnet.ru

Новосибирск, «Топ-книга»,
тел. (3832) 36-10-26, факс 36-10-27.
E-mail: office@top-kniga.ru
http://www.top-kniga.ru

Тюмень, «Друг»,
тел./факс (3452) 21-34-82.
E-mail: drug@tyumen.ru

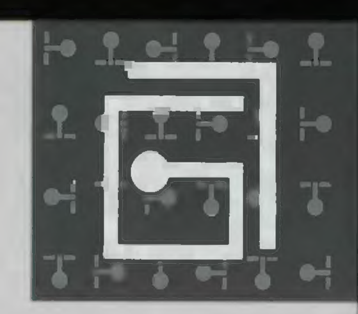
Тюмень, «Фолиант»,
тел. (3452) 27-36-06, факс 27-36-11.
E-mail: foliant@tyumen.ru

Челябинск, ТД «Эврика», ул. Барбюса, д. 61,
тел./факс (3512) 52-49-23.
E-mail:evrika@chel.surnet.ru

**Татарстан**
Казань, «Таис»,
тел. (8432) 72-34-55, факс 72-27-82.
E-mail: tais@bancorp.ru

**Урал**
Екатеринбург, магазин № 14,
ул. Челюскинцев, д. 23,
тел./факс (3432) 53-24-90.
E-mail: gvardia@mail.ur.ru

Екатеринбург, «Валео-книга»,
ул. Ключевская, д. 5,
тел./факс (3432) 42-56-00.
E-mail: valeo@etel.ru



# Ассемблер для процессоров Intel Pentium

Значение языка ассемблера трудно переоценить. Все без исключения средства разработки программ в той или иной степени используют ассемблер. К примеру, большинство библиотечных функций, входящих в Visual C++ и Delphi и составляющих их основу, написаны на ассемблере. Мультимедийные приложения, программы обработки сигналов и многие другие используют высокопроизводительные библиотеки функций, разработанные с помощью ассемблерных команд технологии SIMD. Большинство приложений, работающих в режиме реального времени, либо написаны целиком на ассемблере, либо используют в критических участках кода ассемблерный код. Изучение современного ассемблера — задача далеко не простая, и эта книга, расширенное руководство по применению ассемблера процессоров Intel Pentium, позволит читателю успешно ее решить. Для опытных программистов она будет полезна в качестве справочного пособия, так как содержит много справочной информации по командам ассемблера и современным технологиям обработки данных. Не являясь учебником, она может использоваться и в этом качестве теми, кто хотел бы изучить ассемблер самостоятельно.

Тема: **Программирование/Assembler** Уровень читателя: **начинающий/опытный**




197198, Санкт-Петербург, а/я 619
тел.: (812) 703-73-74, postbook@piter.com

61093, Харьков-93, а/я 9130
тел.: (057) 712-27-05, piter@kharkov.piter.com

